Object Oriented Programming

Module

Information Systems

2023

**Table of Contents**

[**Chapter 1: Introduction to Object-Oriented Programming (OOP)** 3](#_Toc125621901)

[**1.1.** **Overview of OOP?** 3](#_Toc125621902)

[**1.2.** **Why Java?** 4](#_Toc125621903)

[**1.3.** **The JVM and Byte Code** 5](#_Toc125621904)

[**1.3.1.** **Java Technology** 5](#_Toc125621905)

[**1.3.2.** **Phases of Java Program** 6](#_Toc125621906)

[**1.4.** **Basic concepts of OOP** 7](#_Toc125621907)

[**1.5.** **Basics in Java** 12](#_Toc125621908)

[**Chapter 2: The inside of objects and classes: More on OOP concepts** 14](#_Toc125621909)

[**2.1.** **Member methods and their components** 14](#_Toc125621910)

[**2.2.** **Classes and Object** 15](#_Toc125621911)

[**2.3.** **Constructors** 20](#_Toc125621912)

[**2.4.** **Static and Instance members** 22](#_Toc125621913)

[**2.5.** **Member Methods** 24](#_Toc125621914)

[**2.6.** **Access Specifiers / Access Modifiers** 27](#_Toc125621915)

[**2.7.** **Accessors and Mutators/ Getters and Setters Methods** 29](#_Toc125621916)

[**3.** **Chapter 3: Inheritance** 30](#_Toc125621917)

[**3.1.** **Concept of inheritance** 32](#_Toc125621918)

[**3.2.** **Using this() and super()** 35](#_Toc125621919)

[**3.3.** **Final Classes, Abstract Classes and Interfaces** 39](#_Toc125621920)

[**3.3.1.** **Final Classes and Methods** 39](#_Toc125621921)

[**3.3.2.** **Abstracts classes and methods** 41](#_Toc125621922)

[**3.3.3.** **Interfaces** 44](#_Toc125621923)

[**3.4.** **Package** 47](#_Toc125621924)

[4. Chapter 4: Polymorphism 49](#_Toc125621925)

[**4.1.** **Polymorphism with Overloading and Overriding** 52](#_Toc125621926)

[**5.** **Chapter 5: Exception Handling** 54](#_Toc125621927)

[**5.1.** **Exception handling overview** 54](#_Toc125621928)

[**5.1.1.** **The causes of exceptions** 54](#_Toc125621929)

[**5.1.2.** **The Throwable class hierarchy** 55](#_Toc125621930)

[**5.1.3.** **Handling of an exception** 56](#_Toc125621931)

[**5.2.** **User defined exceptions** 62](#_Toc125621932)

[**6.** **Chapter 6: Files and Streams** 63](#_Toc125621933)

[**6.1.** **Introduction** 63](#_Toc125621934)

[**6.2.** **STREAM** 64](#_Toc125621935)

[**6.3.** **Types of Stream** 65](#_Toc125621936)

[**6.4.** **I/O classes** 66](#_Toc125621937)

[**6.4.1.** **Byte Based Stream** 66](#_Toc125621938)

[**6.4.2.** **Character based Streams** 73](#_Toc125621939)

[**6.5.** **The File Class** 76](#_Toc125621940)

[**6.6.** **Predefined Streams** 78](#_Toc125621941)

[**Reference** 80](#_Toc125621942)

# **Chapter 1: Introduction to Object-Oriented Programming (OOP)**

Computer *programs*, known as *software*, are instructions to the computer. We tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so we need to use computer languages to communicate with them. Programs are written using programming languages.

* **Programming Languages**
* **Machine language** is a set of primitive instructions built into every computer. The instructions are in the form of binary code, so you have to enter binary codes for various instructions. Program with native machine language is a tedious process. Moreover the programs are highly difficult to read and modify. For example, to add two numbers, you might write an instruction in binary like this: **1101101010011010**
* **Assembly languages** were developed to make programming easy. Since the computer cannot understand assembly language, however, a program called assembler is used to convert assembly language programs into machine code. For example, to add two numbers, you might write an instruction in assembly code like this:

ADDF3 R1, R2, R3

* **The high-level languages** are English-like and easy to learn and program. For example, the following is a high-level language statement that computes the area of a circle with radius 5: area = 5 \* 5 \* 3.1415;
* Popular High-level language include: COBOL (COmmon Business Oriented Language), FORTRAN (FORmula TRANslation), BASIC (Beginner All-purpose Symbolic Instructional Code), Pascal (named for Blaise Pascal), Ada (named for Ada Lovelace), C (whose developer designed B first), Visual Basic (Basic-like visual language developed by Microsoft), Delphi (Pascal-like visual language developed by Borland), C++ (an object-oriented language, based on C), **Java.**

## **Overview of OOP?**

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code that manipulates that data. OOP is a way of organizing and structuring code in a way that is easy to understand, maintain, and expand upon. It is one of the most popular programming paradigms in use today, and is the basis for many modern programming languages, such as Java, C++, and Python. OOP is based on the idea that objects, which are instances of classes, interact with one another to perform tasks. Each object has its own data and behavior, and can interact with other objects through methods and properties. OOP promotes code reusability, encapsulation, and modularity, which means that the code is divided into small, manageable chunks, making it easier to understand, maintain, and update.

**Object-oriented programming (OOP) in Java**

Java is an object-oriented programming language that is widely used for developing a wide range of applications, from desktop and mobile applications to web services and enterprise systems. The Java language is built on the principles of OOP, which means that it provides a framework for creating and manipulating objects.

In Java, a class is a blueprint for creating objects, and an object is an instance of a class. Each class has a set of properties (also known as fields or variables) that define the data that an object can hold, and a set of methods (also known as functions) that define the behavior of an object. Java provides a rich set of keywords and constructs that support the basic OOP concepts of encapsulation, inheritance, polymorphism and interfaces.

Encapsulation in Java allows to hide the implementation details of a class from other objects, by using the private keyword to restrict access to the fields of a class. Java also provides a mechanism of inheritance which allows a new class to inherit properties and methods from an existing class, by using the extends keyword. Polymorphism allows objects of different classes to be treated as objects of a common superclass, using the "is-a" relationship. And interfaces allows a class to inherit the methods from multiple super classes.

In summary, Java is a powerful object-oriented programming language that provides a rich set of features for developing robust, maintainable, and scalable applications. Its OOP constructs make it easy to model real-world concepts and to build complex systems with minimal effort, and its wide range of libraries and frameworks make it easy to perform common tasks and to integrate with other systems.

## **Why Java?**

Java technology is more than just a programming language. As a programming language, Java can create all kinds of applications that you could create using any conventional programming language. Java technology applications are typically general-purpose programs that run on any machine where the Java runtime environment (JRE) is installed. Java enables users to develop and deploy applications on the

* + Desktop computers…………..Standalone Application
  + Internet for servers …………..Servlets, Applets
  + Small hand-held devices ……. Mobile Application

## **The JVM and Byte Code**

### **Java Technology**

Java Technology mainly consists two parts

* + JDK (Java Development Kit)
  + JRE (Java Runtime Environment)

1. **JDK (Java Development Kit)**: JDK is A free software development package from Sun Microsystems that implements the basic set of tools needed to write, test and debug Java applications and applets.
   * i.e. JDK is Java Software Development kit (SDK)

It is the **toolkit** for developers that includes the Java compiler and the runtime environment. To write Java programs, you need the JDK installed. The major component of JDK are

* + **Java compiler** (.java file 🡪 .class file)
  + **Java launcher** and **jdb debugger**

**JDK Editions:**

* **Java Standard Edition (J2SE)**
  + J2SE can be used to develop client-side standalone *applications* or *applets*.
  + **Java Enterprise Edition (J2EE)**
* J2EE can be used to develop server-side applications such as *Java servlets* and Java *Server Pages*.
  + **Java Micro Edition (J2ME)**
    - J2ME can be used to develop applications for *mobile devices* such as *cell phones*.

1. **JRE (Java Runtime Environment)**: **JRE** is the program that emulates the **JVM**, so that users can run Java programs. To run Java programs, you need download and install the JRE. Runs code compiled for a JVM and performs class loading (through the class loader), code verification (through the bytecode verifier) and finally code execution. The major component of JRE are

* **Java platform core classes libraries**
* **Java virtual machine (JVM)**

### **Phases of Java Program**

Java programs normally go through five phases – ***edit, compile, load, verify*** and***execute***

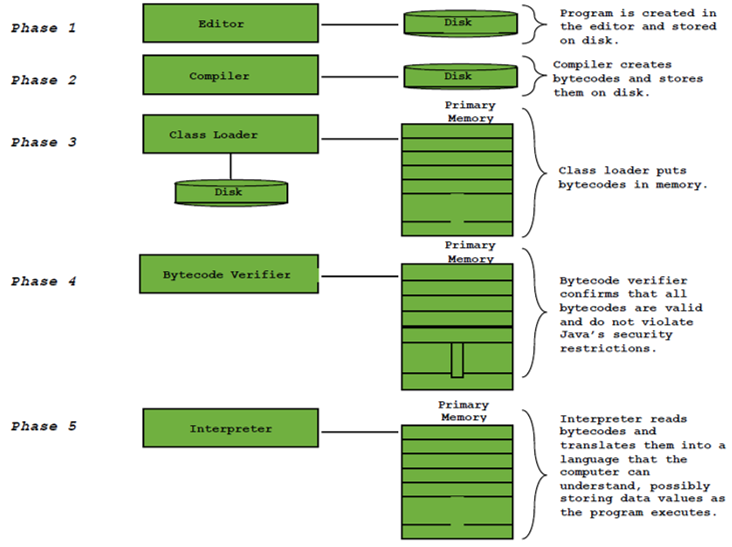


Figure 1 Phases of Java Programs

1. **Creating a Program**: Involves editing a file with an editor. The following are some Java Integrated Development Environments (IDEs). Jbuilder, NetBeans , Sun ONE Studio, Eclipse, jEdit, Jcreator, BlueJ, jGRASP, etc. Most of this editors can be downloaded for free. Any text editor can also be used for creating Java programs. Programs written on IDEs or any other editors are known as Source Code. Java source code file names end with .java extension
2. **Compiling Java Programs into Bytecodes:** Since a computer cannot understand a source program, program called a **compiler** is used to translate the source program into a machine language program called an **object program**.With Java, you write the program once, and compile the source program into a special type of object code, known as **bytecode**. The **bytecode** can then run on any computer (platform) with a Java Virtual Machine. A **virtual machine** is a software application that simulates a computer-JVM is one of the most widely used virtual machine. Java Virtual Machine, which is part of JRE, is a software that interprets Java bytecode. To compile Java Program ( ***Javac welcome.java );*** To execute java programs (***Java welcome)***
3. **Loading a Program**: A program must be placed in memory before it can execute: a process known as **loading**. The class loader takes the .class file (produced during compilation process) containing the program’s bytecodes and transform them to primary memory. The class loader also loads any of the .class files provided by java that your program uses. The .class files can be loaded from a disk on your system or over a network.
4. **Bytecode Verification**: Involves examining bytecodes to ensure that they are valid and do not violate **Java’s security restriction**. Java enforces strong security, to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might)
5. **Execution**: The JVM executes the program’s bytecodes, thus performing the actions specified by the program. “**.class**” file is not final machine code yet. It is virtual machine code. JVM need a second compilation to convert virtual machine code to real machine code. We call it just-in-time compiler (JIT). This greatly affects the speed of the execution. Traditionally, our C/C++ program can finish all the compilation before execution. The generated machine code could run directly on OS or hardware

## **Basic concepts of OOP**

The basic concepts of Object-Oriented Programming (OOP) in Java include:

**Classes and Objects**: Classes are templates for creating objects, and objects are instances of classes.

Encapsulation: Encapsulation is the process of hiding the implementation details of a class from other objects.

Inheritance: Inheritance is the mechanism that allows a new class to inherit properties and methods from an existing class.

**Polymorphism:** Polymorphism allows objects of different classes to be treated as objects of a common superclass.

Abstraction: Abstraction is the process of hiding the complexity of a system by exposing only the necessary details to the user.

**Interface:** Java interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods.

All these concepts are fundamental to the Java programming language and are implemented through the use of keywords such as "class," "extends," "implements," and "abstract."

**Classes and Objects**

Classes and objects are the fundamental concepts of object-oriented programming (OOP) in Java. A class is a blueprint or template for creating objects, while an object is an instance of a class.A class defines the properties and methods that an object of that class can have, and can also define the behavior of the object. Properties, also known as fields or variables, are used to store the data associated with an object. Methods, also known as functions, are used to define the behavior of an object and to perform operations on the object's data.

In Java, a class is defined using the "class" keyword, followed by the name of the class. The properties and methods of a class are defined inside the class definition, between the curly braces. Properties are defined using the data types (int, String, boolean, etc.) and methods are defined using the return type and method name.

Creating an object in Java is done by using the "new" keyword, followed by the name of the class and parentheses. This creates a new instance of the class, and the object can then be assigned to a variable.

For example, consider a class called "Car", which has properties such as "make" and "model" and methods such as "drive" and "stop". The class definition might look like this:

|  |
| --- |
| class Car {  private String make;  private String model;  public void drive() { // code to drive the car }  public void stop() { // code to stop the car }  } |

To create an object of the class "Car", you would use the following code:

Car myCar = new Car ();

Once an object is created, you can access its properties and methods by using the dot notation. for example, myCar.make = "Toyota"; and myCar.drive()

In summary, classes and objects are the building blocks of object-oriented programming in Java. A class defines the properties and methods that an object can have, and an object is an instance of a class that has its own data and behavior. Classes and objects are used to model real-world concepts and to create complex systems that are easy to understand, maintain, and expand upon.

**Encapsulation**

Encapsulation is one of the fundamental concepts of object-oriented programming (OOP) in Java. It is the process of hiding the implementation details of a class from other objects, and providing a public interface for interacting with the object.

In Java, encapsulation is achieved by using the "private" and "public" access modifiers. Variables and methods that are declared as "private" can only be accessed within the class, while variables and methods that are declared as "public" can be accessed from any other class. This allows for the implementation details of a class to be hidden from other objects, while still providing a way for other objects to interact with the class through its public methods.

Encapsulation is used to protect the integrity of an object's data and to ensure that the object's behavior is consistent. By hiding the implementation details of a class, you can change the internal workings of the class without affecting other objects that use the class. This makes the code more maintainable and less prone to errors.

**Inheritance**

Inheritance is another fundamental concept of object-oriented programming (OOP) in Java. It is a mechanism that allows a new class to inherit properties and methods from an existing class. This allows for code reusability, as a new class can reuse the properties and methods of an existing class, and also allows for a natural hierarchy to be established among classes.

In Java, a class can inherit from another class by using the "extends" keyword. The class that is inheriting is called the subclass, and the class that is being inherited from is called the superclass. A subclass inherits all of the properties and methods of its superclass, and can also have its own properties and methods.

Inheritance is used to establish a "is-a" relationship between classes. For example, a "Car" class can inherit from a "Vehicle" class, because a car is a type of vehicle. The "Vehicle" class can have properties and methods that are common to all vehicles, such as "speed" and "accelerate", and the "Car" class can have properties and methods that are specific to cars, such as "make" and "model".

|  |
| --- |
| **Example**:  Copy code  **class Vehicle** {  private int speed;  public void accelerate(int amount) { speed += amount; }  }  **class Car extends Vehicle** {  private String make;  private String model;  public void honk() { // code to make the car honk }  } |

In this example, the "Car" class inherits from the "Vehicle" class. It inherits the "speed" variable and "accelerate" method from the "Vehicle" class, and also has its own "make" and "model" variables and "honk" method.

In summary, inheritance is a powerful feature of OOP in Java that allows for code reusability and the creation of a natural hierarchy among classes. It establishes a "is-a" relationship between classes and allows a new class to inherit properties and methods from an existing class, which makes it easier to understand, maintain, and expand upon the code.

* **A Simple Java Program**

Every programing starts with a simple set of instruction. The following code is a java program that prints the string “ *Welcome to Java”.* Now lets make a line by line investigation of the program

|  |
| --- |
| 1. **package welcome** 2. **/\* Your first Java program; Welcome.java** 3. **The program prints a string Welcome to Java Programming** 4. **\*/** 5. ***public class Welcome {*** 6. **//main method begins execution of Java application** 7. ***public static void main (String args[])*** 8. ***{*** 9. ***System.out.println(“Welcome to Java”);*** 10. ***}*//end of main method** 11. ***}* //end of class Welcome** |

* ***Line 1. Package***: Grouping of related classes called *packages.*
  + Group of all packages known as Java class library or Java applications programming interface (Java API).
* ***Lines 2-4 & 6 comment lines***: There are two types of comments in Java
  + End-of-line (Single line) comments: A line that begins with **//**
  + Traditional (Multiple line ) Comments: A text that begins in **/\*** ends in **\*/**
    - All text between these delimiters is comment so it is ignored by the compiler.
    - These comment can be applied for a line, more than a lines or part(s) of a line
  + Javadoc comments: A text that begins in /\*\* and ends in\*/
    - As with traditional comments, all text between the Javadoc comment delimiters is ignored by the compiler.
    - Javadoc comments enable programmers to embed program documentation directly in their programs.
* **Line 4: *public class Welcome {***
  + This line begins class declaration of the class Welcome. Every program in Java consists of at least one class declaration. *Generally, Java class decleration has the following format:*
* **[Access level Specifier] class *class\_name* { … }**
* **Access level Specifier :** Could be omitted (none), **public**, **private,** or **protected.** These are keywords that help to set the ***visibility*** and ***accessibility***of a class, its member variables, and methods. They determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class.
* ***class***: The class keyword introduces a class declaration in Java and is immediately followed by the class name (Welcome).
* ***class\_Name:*** Is the name of a class. A Java class name is an identifier. By convention class names should start with capital letters. Java is case Sensitive
* ***Line 7: public static void main(String args[])***: Is a method of which the starting point of every java application. Every Java application must contain one main method. public is a keyword which specifies access level of the method. The JVM executes the application by invoking the **main** method
* ***Line 9: System.out.println( "Welcome*** ***to Java Programming!" );*** This line prints the string contained in quotation. **System.out** is known as the standard output object. When ***System.out.println*** completes its task, it positions the output cursor (the location where the next character will be displayed) to the beginning of the next line while **System.out.print** don’t.
* Every statement of java must end in semicolon (;) so is line 9
* ***Lines 10 and 11***: This line depicts the end of the main method and the class welcome.

## **Basics in Java**

* + 1. **Identifiers:** An identifier is a sequence of characters that consist of letters, digits, underscores (\_), and dollar signs ($). An identifier must start with a letter, an underscore (\_), or a dollar sign ($). It cannot start with a digit. An identifier cannot be a reserved word (keywords). And it can be of any length.

Since Java is case-sensitive, X and x are different identifiers. Identifiers are used for naming variables, constants, methods, classes, and packages. Descriptive identifiers make programs easy to read.

* + 1. **Variables:** To put a value in memory, and later to get it back, a program must have a name for each section of memory that it uses. Variable is a name for a location in main memory which uses a particular data type to hold a value. Variables are used to store data in a program. When new value is placed into a variable, replaces (and destroys) previous value
* **Variable Declaration:** tells the compiler to allocate appropriate memory space for the variable based on its data type.

|  |
| --- |
| The syntax for declaring a variable is: *datatype* variableName*;*  Example: int num; //declared as an integer number  double radius; // declared to be a double number  char a; // declared to be a character  if variables are of the same type, they can be declared together, as follows:  Examples: int i, j, k; |

* **Assignment Statements:** After a variable is declared, you can assign a value to it by using an assignment statement. In java, the equal sign(=) is used as the assignment operator.

|  |
| --- |
| The syntax for assignment statement is as follows: variable = expression; x = 5\*(3/2) + 2; // Assign the value of expression to x  radius = 1.0; // Assign 1.0 to radius;  a = 'A'; // Assign 'A' to a; |

* **Variable Declaration and Initialization:** Declaring and Initializing can be done in one step

|  |
| --- |
| int x = 1  String str = “Hi”;  boolean b = (1 > 2); |

* **Constants:** The values of constants can never be changed once they are initialized. Constant variables are also called named constants or read-only variables.

|  |
| --- |
| The general format for declaration of constants is given by:  final datatype CONSTANTNAME = VALUE;  Examples: final double PI = 3.14159;  final int MAX\_SIZE = 3; |

* + 1. **Keywords:** Are reserved for use by Java and are always spelled with all lowercase letters.

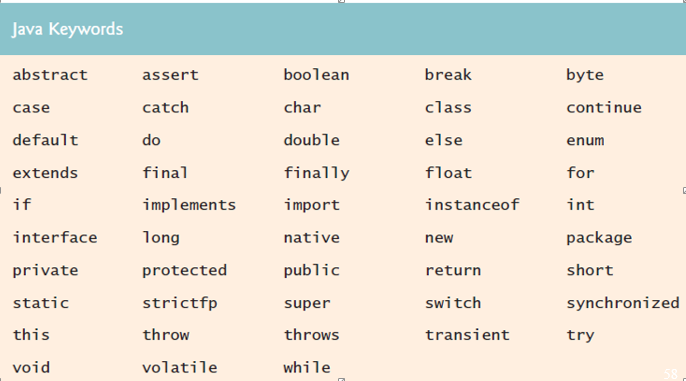


Figure 2 Java Keywords

* + 1. **Primitive Data-types: There are around 8 primitive data types in java. These are**

|  |
| --- |
| **byte, short, int, long, float, double, Boolean, char** |

**Numeric Data Type**

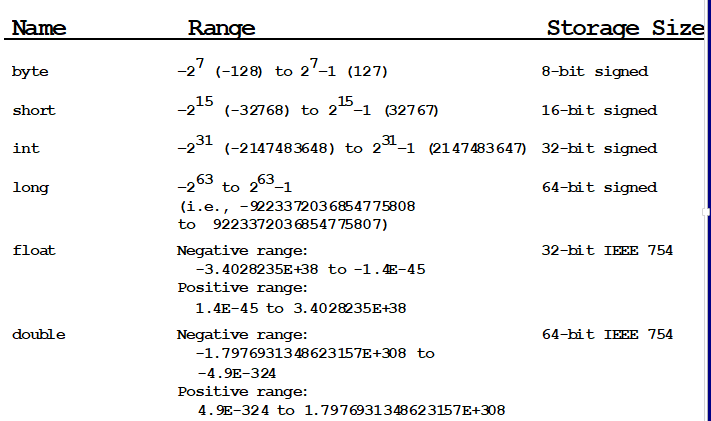


Figure 3 Numberic Data Type their capable range and their storage size

* Java uses four types for integer: byte, shortint, and long.
* Java uses two types for *floating-poin*t numbers: float and double. The double type is twice a big as float.
* The character data type, char, is used to represent a single character. A character literal is enclosed in single quotation marks. (e.g. char letter = ‘A; )
  + 1. **The String Type:** To represent a string of characters, use the data type called *String*.

|  |
| --- |
| **Exampe: *String* message = “Welcome to Java”;** |

# **Chapter 2: The inside of objects and classes: More on OOP concepts**

## **Member methods and their components**

A method is a collection of statements that are grouped together to perform an operation.

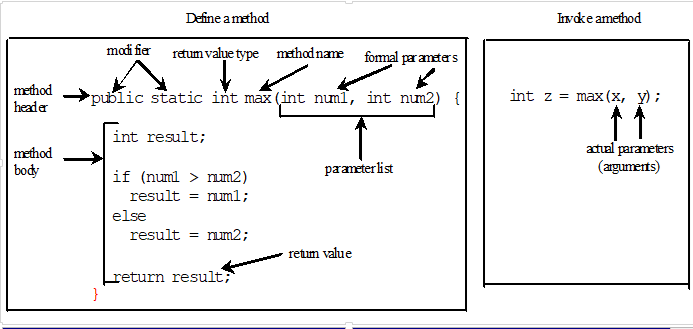


Figure 4 Method Definition

Method signature is the combination of the method name and the parameter list. The variables defined in the method header are known as formal parameters or simply parameter. You need to declare a separate data type for each parameter.

For instance, **int num1, num2** should be replaced by **int num1, int num2**.

When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. A method may return a value. The returnValueType is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword void. For example, the returnValueType in the main method is void.

This program demonstrates calling a method max to return the largest of the **int** values

|  |
| --- |
| **public class TestMax {**  **/\*\* Main method \*/**  public static void main(String[] args) {  int i = 5;  int j = 2;  int k = max(i, j);  System.out.println("The maximum between " + i + “  and " + j + " is " + k);  }  /\*\* Return the max between two numbers \*/  public static int max(int num1, int num2) {  int result;  if (num1 > num2)  result = num1;  else  result = num2;  return result;  }  } |

Calling a method:

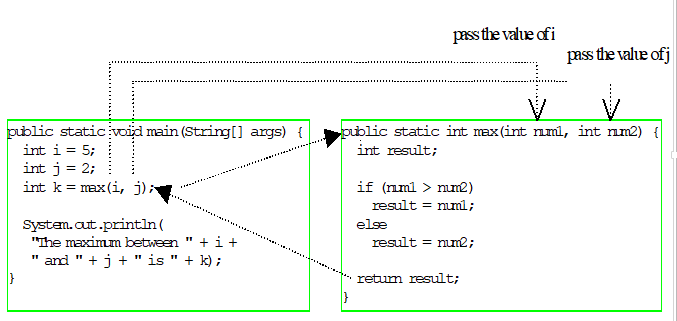


Figure 5 calling a method

## **Classes and Object**

In procedural programming, data and operations on the data are separate. Object-oriented programming places data and the operation that pertain to them within a single entity called an Object. The object-oriented programming approach organizes in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Object-oriented programming (OOP) involves programming using objects.

Using objects improves software reusability and makes program easier to develop and easier to maintain. Programming in java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

**An Object:**  represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a ***unique******identity, state, and behaviors.*** The *state* defines the object, and the *behavior* defines what the object does.

* The ***state*** of an object consists of a set of ***data fields*** (also known as *properties*) with their current values.
* The ***behavior*** of an object is defined by a set of methods.

**Example**: A circle object has a data filed, ***radius***, which is the property that characterizes a circle. One behavior of a circle is that its area can be computed using the method ***getArea().***

Objects of the same type are defined using a common **class**.

**A Class:**  is a blueprint, template, or prototype that defines and describes the **static attributes** and **dynamic behaviors** common to all objects of the same kind. Class is objects with the same attributes and behavior. A class can be visualized as a three-compartment box:

* + ***Name*** (or identity): identifies the class.
  + ***Variables*** (or attribute, state, field): contains the *static attributes* of the class.
  + ***Methods*** (or behaviour, function, operation): contains the *dynamic behaviours* of the class.

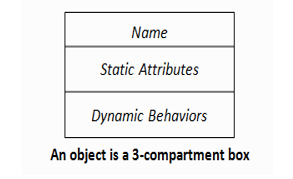


Figure 6 Class

* In object-oriented programming, a **class** is a programming language construct that is used as a blueprint to create objects of that class.
  + - This blueprint describes what an object’s data and methods will be.
    - An object is an instance of a class. The class that contains that instance can be considered as the type of that object.

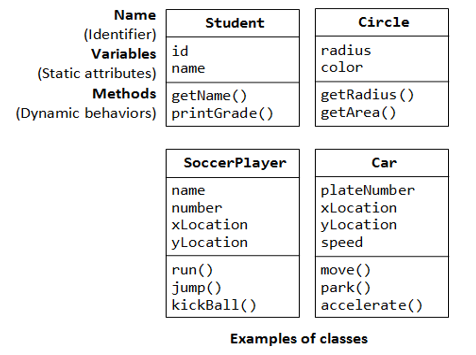


Figure 7 Example of Classes

* An **instance** is a realization of a particular item of a **class**. In other words, an instance is an instantiation of a class.
* All the instances of a class have similar properties, as described in the class definition.
* **For example**, you can define a class called "Student" and create two instances of the class "Student" for "Peter", and "Paul“.
* The term "object" is often used loosely, which may refer to a **class** or an **instance**.
* Each object within a class retains its own states and behaviors.
* The relationship between classes and object is analogous to the relationship between apple recipes and apple pie.
* A single class can be used to instantiate multiple objects. This means that we can have many active objects or instances of a class.
* A class usually represent a noun, such as a person, place or (possibly quite abstract) thing - it is a model of a concept within a computer program.

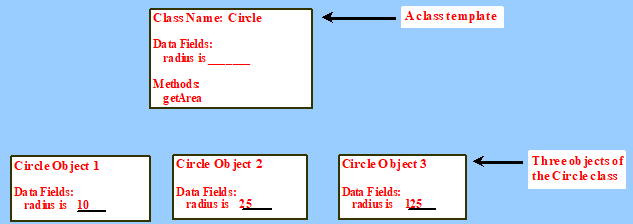


Figure 8 Example: Instances of class Circle

* A Java class uses *variables* to define *data fields* and *methods* to define *behaviors*.
* Variables and methods are called members of the class.
* Additionally, a class provides a special type of methods, known as ***constructors***, which are invoked to construct new objects from the class.

|  |  |
| --- | --- |
| Syntax:  # Class declaration has the following syntax  **[ClassModifiers]class ClassName [pedigree]**  **{**  **//Class body;**  **}**  # Class Body may contain   * *Data fields*   **[FieldModifier(s)] returnType varName ;**   * *Constructors*   **[Modifiers] ClassName(…){…}**   * *Member Methods*   **[MethodModifier(s)] returnType methodName(…){…}**   * And even other *classes* | Example:  Class Circle{  double radius;// member datafields;  //Member methods  double getArea ( )  {  return Math.PI\*radius\*radius;  }  }//class end |

1. **Creation of Objects using Class name**

* Objects are created from classes using ***new*** operator

|  |
| --- |
| ClassName objectRefVar = **new** ClassName( );  OR  ClassName objectRefVar ;  objectRefVar = **new** ClassName( ); |

The **new** operator creates an object of type ClassName. The variable **objectRefVar** references the object . An object can be created without using reference variables.

|  |
| --- |
| new ClassName( ); // object creation without reference variable |

* **Accessing Objects**

Once objects have been created, its data fields can be accessed and its methods invoked using the ***dot operator*(.),** also known as the ***object member access operator***

|  |
| --- |
| * Referencing the object’s data:   *objectRefVar.data*  e.g., myCircle.radius = 100;   * Invoking the object’s method:   *objectRefVar.methodName(arguments)*  e.g., double area = myCircle.getArea(); |

* **Variables of Primitive datatypes and Object-types**
* Variables of a primitive datatypes are created when the variable is declared and initialized when the variable is assigned using the assignment operator ( = ) , where Object type variables are a reference variables and starts referencing an object when “new” operator is used in the declaration of an object and initialized with a default value when the object is created (using Constructors).
* Variables of a Primitive datatype assigns its value to another primitive datatype when an assignment operator is used to relate both as shown in the figure below ( i =j now i will have the same value as j), where when using assignment operator on Object type variable (reference variable) as shown in the figure below (c1=c2 **c1** points to the same object referenced by **c2**. The object previously referenced by **c1** is no longer referenced it becomes a candidate **for automatic Garbage Collection**, the object is known as **garbage**. Garbage is automatically collected by JVM.)

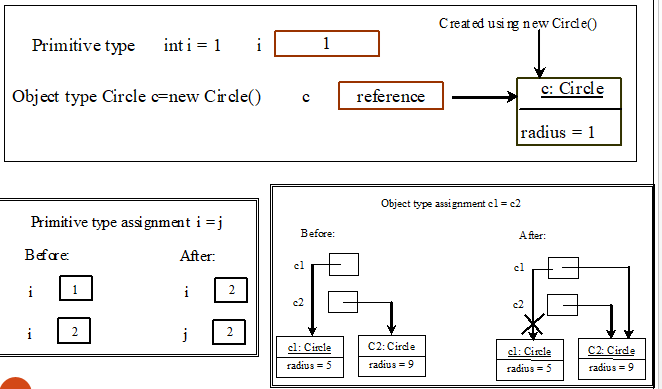


Figure 9 Primitive type variables VS Object type Variables

## **Constructors**

Constructor is a special method that gets invoked “automatically” at the time of object creation. A *constructor* can perform any action, but constructors are designed to perform initializing actions, such as initializing the data field of objects. Constructor is normally used for initializing objects with default values unless different values are supplied. When objects are created, the initial value of data fields is unknown unless its users explicitly do so.

|  |
| --- |
| Syntax:  class Classname{  //declaration of datafields;  **Classname ( ){**  **//Intialization of Insatance variables**  **}**  } |

Constructors has exactly the same name as the defining class. Like regular methods, constructor can be overloaded.(i.e. A class can have more than one constructor as long as they have different signature (different input arguments syntax). This makes it easy to construct objects with different initial data values.

* **Default constructor:** If you don't define a constructor for a class, a *default parameterless constructor* is automatically created by the compiler. The default constructor calls the default parent constructor (super()) and initializes all instance variables to default value (*zero for numeric types, null for object references, and false for Booleans*).(Inheritance…will see it later )

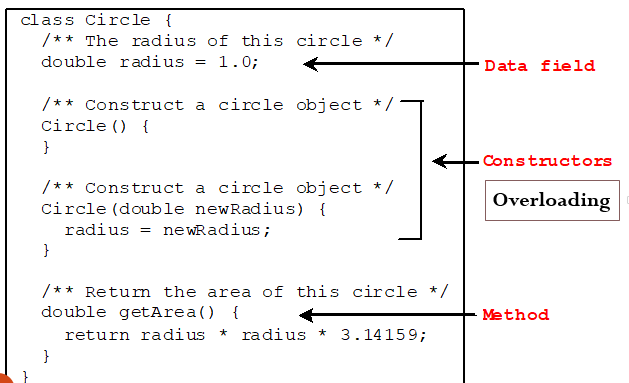


Figure 10 Example of Constructor

* **Overloaded Constructors:** it is possible to define multiple constructors in one class which have different number or different types of parameters that is called Overloading.

|  |
| --- |
| **Creation of Object using Constructors**  ClassName objectRefVar = **new** ClassName([Parameters ]);  OR  ClassName objectRefVar ;  objectRefVar = **new** ClassName([Parameters ]);  OR  new ClassName([Parameters ]) |

The object may be created using any of its constructors. If it has constructors other than the default one appropriate parameters must be provided to create objects using them.

|  |
| --- |
| **Example:**  **/\*Declare a reference variable and create an object using an empty constructor \*/**  **Circle c1 = new Circle();**  **/\*Declare a reference variable and create an object using constructor with an argument \*/**  **Circle c2 = new Circle(5.0)**  **//Declare reference variables**  **Circle c3, c4 ;**  **/\*Create objects and refer the objects by the reference variables**  **c3 = new Circle();**  **c4 = new Circle(8.0);** |

* **Differences between methods and constructors**.
  + Constructors must have the same name as the class itself.
  + Constructors do not have a return type—not even **void.**
  + Constructors are invoked using the ***new*** operator when an object is created.

## **Static and Instance members**

* **Data fields:** Data Fields are variables declared directly inside a class (not inside method or constructor of a class). Data fields can be variables of primitive types or reference types.

|  |  |
| --- | --- |
| public class Student {  String name;  int age;  boolean isSci;  char gender;  } | public class Test {  public static void main(String[] args) {  Student student = new Student();  System.out.println("name? " + **student.name**);  System.out.println("age? " + **student.age**);  System.out.println(" isSci? " + **student.isSci**);  System.out.println("gender? " + **student.gender**);  }  } |

The default value of a data field is null for a reference type, 0 for a numeric type, false for a boolean type, and '\u0000' for a char type.

* Data fields can be initialized during their declaration.
* However, the most common way to initialize data fields is inside constructors
* **Types of Variables**

1. **Local variable:** Created inside a method or block of statement. This method below has local variables such as a, b, sum.Their scope is limited to the method.

|  |
| --- |
| E.g.  public int getSum(int a, int b) { int sum=0; sum=a+b; return sum; } |

1. **Instance variable:** Created inside a class but outside any method. Store different data for different objects. The method below has two instance variable year and month.

|  |
| --- |
| E.g.  public class Sample1 { int year; int month; public void setYear(int y){  year=y; } … } |

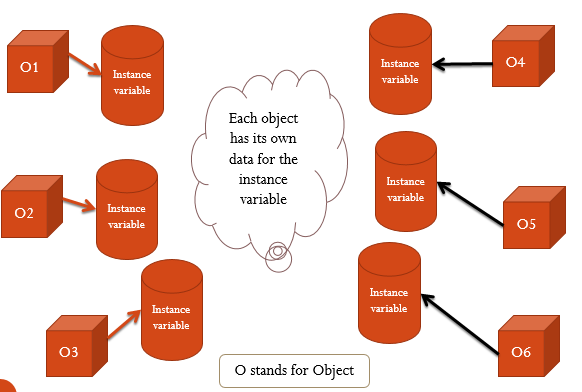


Figure 11 Instance Variable

1. S**tatic variable:** Created inside a class but outside any method. Store one data for all objects/instances. The class below has one static variable.

|  |
| --- |
| E.g.  public class Sample { static int year; int month; public static void setYear(int y) {  year=y; } … } |

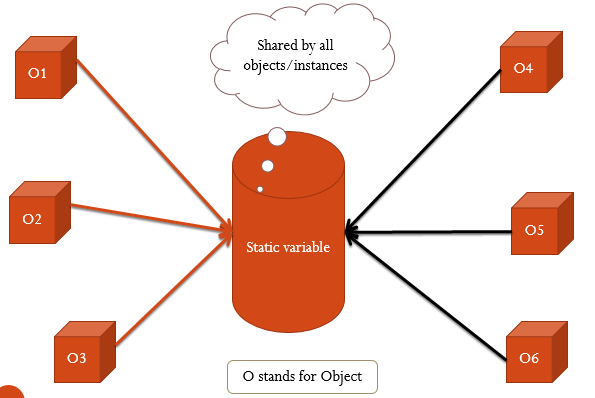


Figure 12 Static Variable

## **Member Methods**

A method declared inside the class definition is a member method. Methods implement the behavior of classes. That is method is an operation which can modify an objects behavior. In other words, it is something that will change an object by manipulating its variables. Only an object's methods should modify its variables

* Instance variables and methods (also called **members**) are those associated with individual instances (or objects) of a class.
* Instance variables belong to a specific instance; it is not shared among objects of the same class.

|  |
| --- |
| **Example:**  Circle cl = new Circle();  Circle c2 = new Circle(5); |

* + The radius in c1 is independent of the radius in c2, and is stored in different memory location. Change c1’s radius do not affect c2’raduis and vice versa.
* Instance methods are invoked by specific instance of the class.
* To get to the value of an instance variable or to invoke instance methods, we use an expression in what's called dot notation.

|  |
| --- |
| **Example:**  Circle cl = new Circle ();  cl.radius = 5;  System.out.println(“Area “ + cl.getArea()); |

**Static/ Class Variables**

* If we want all the instances of a class to share data, use ***static variables***.
* **Static/class variables** store values for the variables in common memory location. Because of this common location, all objects of the same class are affected if one object changes the value of a static variable.
* **With instance variables**, each new instance of the class gets a new copy of the instance variables that class defines.
* Each instance can then change the values of those instance variables without affecting any other instances.
* **With static/class variables**, there is only one copy of that variable. Every instance of the class has access to that variable, but there is only one value. Changing the value of that variable changes it for all the instances of that class.
* We define class variables by including the **static** keyword before the variable itself.

|  |
| --- |
| **Example:**  class FamilyMember {  **static String surname = "Johnson";**  String name;  int age;  ... } |

* Instances of the class **FamilyMember** each have their own values for name and age. But the class variable surname has only one value for all family members. Change surname, and all the instances of FamilyMember are affected.

**Static / Class Methods**

* Class methods, like class variables, apply to the class as a whole and not to its instances.
* Class methods are commonly used for general utility methods that may not operate directly on an instance of that class, but fit with that class conceptually.
* For example, the class method ***Math.max()*** takes two arguments and returns the larger of the two. You don't need to create a new instance of Math; just call the method anywhere you need it, like this:

|  |
| --- |
| int LargerOne = Math.max(x, y); |

Good practice in java is that, static methods should be invoked with using the class name though it can be invoked using an object.

|  |
| --- |
| ClassName.methodName(arguments)  OR  objectName.methodName(arguments) |

General use for java static methods is to access static fields.

**Constant Variables:** Constants in class are shared by all objects of the class. Thus, constants should be declared *final static*.

|  |
| --- |
| Example: The constant PI in the Math class is defined as:  final static double PI = 3.14159265 |

* Static variables and methods can be used from instance or static method in the class.
* Instance variables and methods can only be used from instance methods, not from static methods.
* Static variables and methods belong to the class as a whole and not to particular objects.

|  |
| --- |
| **Example:**  public class Foo{  int i=5;  Static int k = 2;  public static void main(String[] args){  int j = i; // wrong i is instance variable  m1(); //wrong m1() is an instance method  }  public void m1(){  //Correct since instance and static variables and can be used in an  instance method.  i = i + k + m2(int i,int j);  }  public static int m2(int I, int j){  return (int)(Math.pow(i,j));  }  } |

## **Access Specifiers / Access Modifiers**

Java provides a number of access modifiers to help you set the level of access you want for classes as well as the fields, methods and constructors in your classes. Access modifiers are keywords that help set the visibility and accessibility of a class, its member variables, and methods.Determine whether a field or method in a class, can be used or invoked by another method in another class or subclass. Access modifiers determine the visibility or accessibility of an object’s attributes and methods to other objects.

**Access Modifiers: private, protected, no modifier (**also sometimes referred as ‘**package- private**’ or ‘**default**’ or ‘**friendly**’ access**), public**

Modifiers are applied by prefixing the appropriate keyword for the modifier to the declaration of the class, variable, method or constructor. Combinations of modifiers may be used in meaningful ways.

* **Level of Access of Modifiers:** The two levels are *class level access modifiers* and *member level access modifiers*.

1. **Class level access modifiers (java classes only)**

* Only two access modifiers is allowed, *public and no modifier*
* If a class is ‘public’, then it CAN be accessed from ANYWHERE.
* If a class has ‘no modifer’, then it CAN ONLY be accessed from ’same package’.

1. **Member level access modifiers (variables and methods)**

* All the four ***public, private, protected and no modifer*** is allowed.
* ***public* access modifier:** The class, data, or method is visible to any class in any package. Fields, methods and constructors declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package.
* **private access modifier**
* The private (most restrictive) fields or methods can be accessed only by the declaring class.
* Fields, methods or constructors declared private are strictly controlled, which means they cannot be accesses by anywhere outside the enclosing class.
* A standard design strategy is to make all fields private and provide public **getter** and **setter** methods for them to read and modify.
* **protected access modifier**
* The protected fields or methods cannot be used for classes in different package.
* Fields, methods and constructors declared protected in a ***superclass*** can be accessed only by ***subclasses*** in other packages.
* Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member’s class.
* **No modifier (default access modifier):** Java provides a default specifier which is used when no access modifier is present. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Access Modifiers*** | ***Same Class*** | ***Same Package*** | ***Subclass*** | ***Other packages*** |
| **public** | **Y** | **Y** | **Y** | **Y** |
| **protected** | **Y** | **Y** | **Y** | **N** |
| **no access modifier** | **Y** | **Y** | **N** | **N** |
| **private** | **Y** | **N** | **N** | **N** |

Figure 13 Member Level Access

|  |
| --- |
| **Interpretation from the Above class**  First row {public Y Y Y Y} is interpreted as:   * Y:- A member declared with ‘public’ access modifier CAN be accessed by the members of the ’same class’. * Y:- A member declared with ‘public’ access modifier CAN be accessed by the members of the ’same package’. * Y: -A member declared with ‘public’ access modifier CAN be accessed by the members of the ’subclass’. * Y:- A member declared as ‘public’ CAN be accessed from ‘Other packages’.   Second row {protected Y Y Y N} is interpreted as:   * Y:- A member declared with ‘protected’ access modifier CAN be accessed by the members of the ’same class’. * Y:- A member declared with ‘protected’ access modifier CAN be accessed by the members of the ’same package’. * Y: -A member declared with ‘protected’ access modifier CAN be accessed by the members of the ’subclass’. * N:- A member declared as ‘protected’ CAN not be accessed from ‘Other packages’.   Third row { Y Y N N} is interpreted as:   * Y:- A member declared with ‘no access modifier’ access modifier CAN be accessed by the members of the ’same class’. * Y:- A member declared with ‘no access modifier’ access modifier CAN be accessed by the members of the ’same package’. * N: -A member declared with ‘no access modifier’ access modifier **CAN not** be accessed by the members of the ’subclass’. If the subclasses are in different package. * N:- A member declared as ‘no access modifier’ **CAN not** be accessed from ‘Other packages’.   Forth row {private Y N N N} is interpreted as:   * Y:- A member declared with ‘private’ access modifier CAN be accessed by the members of the ’same class’. * N:- A member declared with ‘private’ access modifier **CAN not** be accessed by the members of the ’same package’. * N: -A member declared with ‘private’ access modifier **CAN not** be accessed by the members of the ’subclass’. If the subclasses are in different package. * N:- A member declared as ‘private’ **CAN not** be accessed from ‘Other packages’. |

## **Accessors and Mutators/ Getters and Setters Methods**

In Java accessors are used to get the value of a private field and mutators are used to set the value of a private field. Accessors are also known as getters and mutators are also known as setters. A class’s private fields can be manipulated only by methods of that class.

So a client of an object—that is, any class that calls the object’s methods—calls the class’s public methods to manipulate the private fields of an object of the class. Classes often provide public methods to allow clients of the class to set (i.e., assign values to) or get (i.e., obtain the values of) private instance variables. The names of these methods need not begin with *set or get, but this naming convention* is highly recommended in Java.

|  |
| --- |
| **Example:**  public class Circle {  private double x,y,r;  public double getX() { return x;}  public double getY() { return y;}  public double getR() { return r;}  public void setX(double x\_in) { x = x\_in;}  public void serY(double y\_in) { y = y\_in;}  public void setR(double r\_in) { r = r\_in;}  //Methods to return circumference and area  } |

From the above example, a better way of initialising or access data members x, y, r is to use the get and set methods:

|  |
| --- |
| * To initialise/Update a value   aCircle.setX( 10 )   * To access a value:   aCircle.getX() |

# **Chapter 3: Inheritance**

The three fundamental object oriented programming principles are:

* Encapsulation (data hiding) and Data abstraction
* Inheritance
* Polymorphism

Main purpose of these principles is to manage software system complexity by improving software quality factors.

* **Encapsulation** refers to the combining of fields and methods together in a class such that the methods operate on the data, as opposed to users of the class accessing the fields directly.

It is the technique which involves making the fields in a class private and providing access to the fields via public methods.

* If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.
* For this reason, encapsulation is also referred to as data hiding.
* Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.
* The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code.

|  |  |  |
| --- | --- | --- |
| **Example:**  public class EncapTest{  private String name;  private String idNum;  private int age;  public int getAge(){  return age;  }    public String getName(){  return name;  }  public String getIdNum(){  return idNum;  }  public void setAge( int newAge){  age = newAge;  }    public void setName(String newName){  name = newName;  }    public void setIdNum( String newId){  idNum = newId;  }  } | public class RunEncap{    public static void main(String args[]){  EncapTest encap = new EncapTest();  encap.setName("James");  encap.setAge(20);  encap.setIdNum("12343ms");    System.out.print("Name : " + encap.getName()+ " Age : "+ encap.getAge());  }  }   |  | | --- | | **Out Put: Name : James Age : 20** | |

* The public methods are the access points to this class’s fields from the outside java world.
* Normally these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters.
* The variables of the EncapTest class can be accessed as below:
* **Data Abstraction:** Classes normally hide the details of their implementation from their clients. This is called information hiding. The client cares about what functionality a class offers, not about how that functionality is implemented. This concept is referred to as data abstraction. On the example below the detail of the method is represented by the method name “getFirstName”

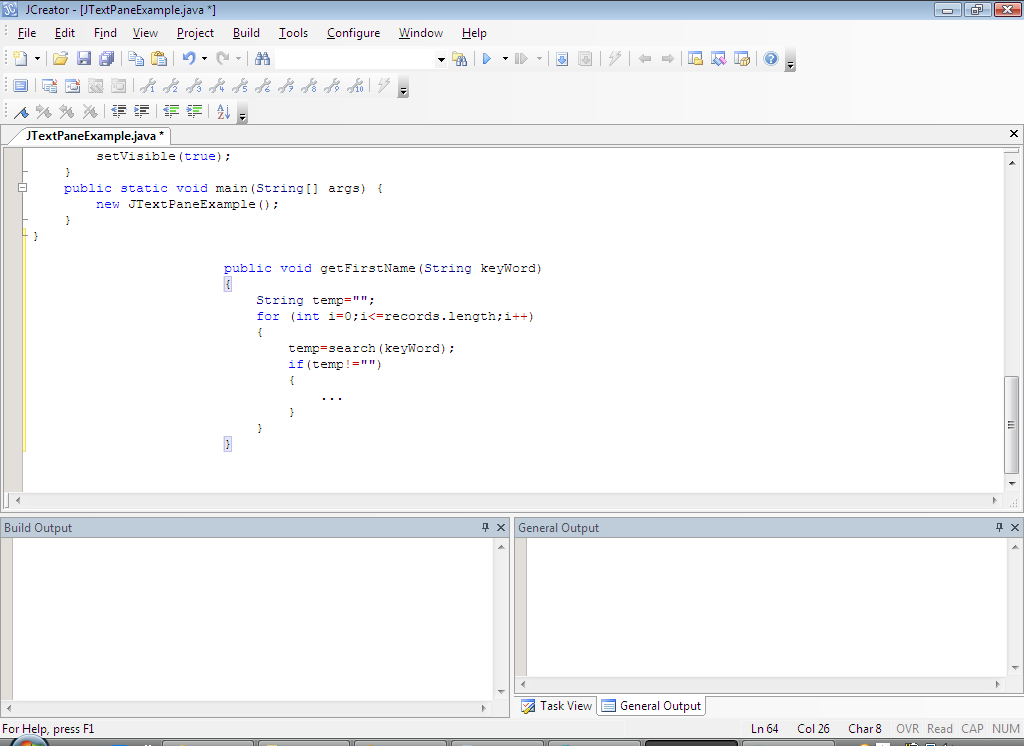


Figure 14 Data abstraction

## **Concept of inheritance**

Object-oriented programming (OOP): allows you to derive new classes from existing classes. This is called inheritance. In other words, Inheritance is a way to form new classes using classes that have already been defined. A class C1 extended from another class C2 is called a subclass, and C2 is called a superclass. A superclass is also refered to as a supertype, a parent class, or a base class and subclass as subtype, a child class, an extended class, or derived class.

* + - * **A subclass:** may inherit accessible data fields and methods from its **superclass** (the immediate parent and all its Ancestors). may add new data fields and methods. may also override an inherited method by providing its own version, or hide an inherited variable by defining a variable of the same.

Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as ***method overriding***. Inheritance is employed to help reuse existing code with little or no modification. The inheritance relationship of **subclasses** and **superclasses** gives rise to a hierarchy. With the use of inheritance the information is made manageable in a hierarchical order.

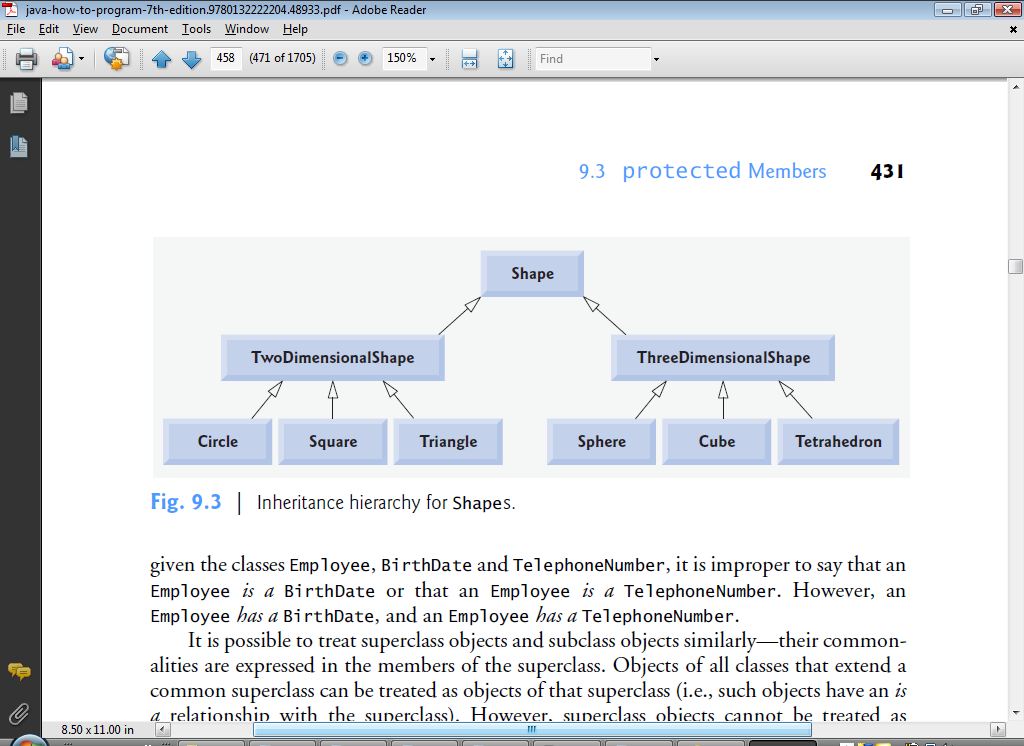


Figure 15 Inheritance Example: Hierarchy of Shape

Elements to be inherited from parent class are protected and public members. Private members of the superclass are not inherited by the subclass and can only be indirectly accessed. Members that have default accessibility in the superclass are also not inherited by subclasses in other packages. These members are only accessible by their simple names in subclasses within the same package as the superclass.

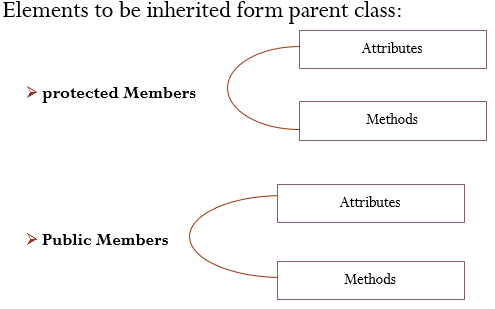


Figure 16 Elements to be inherited

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

* **Applications of inheritance**
* **Specialization:** The new class or object has data or behavior aspects that are not part of the inherited class.
* **Overriding:** Permit a class or object to replace the implementation of an aspect—typically a behavior—that it has inherited
* **Code re-use:** Re-use of code which already existed in another class.

|  |
| --- |
| **In Java inheritance is represented by the key word extends.**  **Syntax**  **modifier class SubclassName extends SuperclassName**  **{**  **//body**  **}**  Example  **public class Mammal extends Animal**  **{**  **//body**  **}** |

The **inheritance** relationship is transitive: if class **x** extends class **y**, then a class **z**, which extends class **x**, will also inherit from class **y.**

|  |
| --- |
| public class Animal{ …}  public class Mammal extends Animal{…}  public class Reptile extends Animal{…}  public class Dog extends Mammal{…} |

* Now based on the above example, In Object Oriented terms following are true:
  + Animal is the superclass of Mammal and Reptile classes.
  + Mammal and Reptile are subclasses of Animal class.
  + Dog is the subclass of both Mammal and Animal classes.
* A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore the following is illegal:

**public class Dog extends Animal, Mammal{…}**

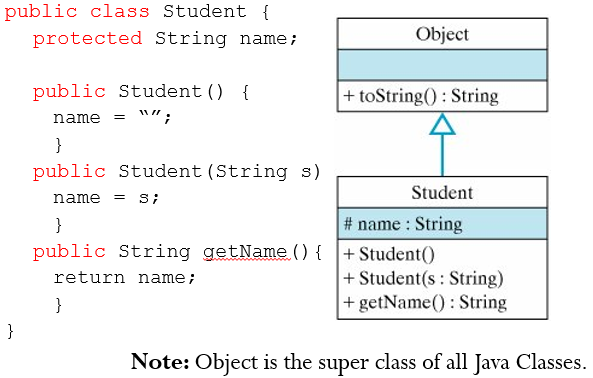


Figure 17 Object class

* All classes in Java by default "extend" the Object class, that's why Object is superclass of every class in Java. Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
* The **instanceof** operator compares an object to a specified type.
* You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.

|  |  |
| --- | --- |
| public class Dog extends Mammal{  public static void main(String args[]){  Animal a = new Animal();  Mammal m = new Mammal();  Dog d = new Dog();  System.out.println(m **instanceof** Animal);  System.out.println(d **instanceof** Mammal);  System.out.println(d **instanceof** Animal);  }  } | Output:  true  true  true |

|  |  |
| --- | --- |
| **Example:**  **class Animal** {  public int numberOfLegs;  public void talk() {  System.out.println("Hello");  }  **class Dog extends Animal** {  public int numberOfFleas;  public Dog() {  numberOfLegs = 4;  numberOfFleas = 10;  }  public void bark() {  System.out.println("Woof woof");  }  public void scratch() {  if (numberOfFleas > 0) numberOfFleas--;  }  } | **Dog d=new Dog();**  d.bark();  d.scratch();  System.out.println(d.numberOfFleas);    - Because the Dog class inherits from the Animal class, we can also use the d reference to access the methods and properties of the Animal class. Use the code below.  d.talk();  System.out.println(d.numberOfLegs); |

## **Using this() and super()**

* **“this” key word:**
* Within an instance method or a constructor, this is a reference to the *current object (*the object whose method or constructor is being called) Use this to refer to the object that invokes the instance

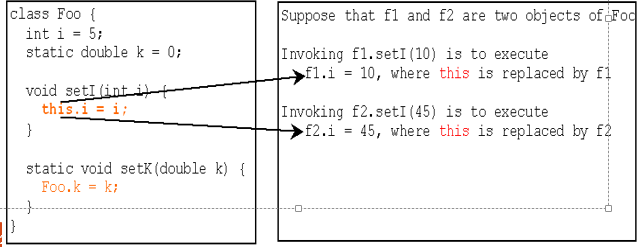


Figure 18 this keyword

* The most common reason for using **this** keyword is because a field is shadowed by a method or constructor parameter. (**to avoid name conflicts**.) Use this to refer to an instance data field.

|  |  |
| --- | --- |
| **For example**, the Point class was written like this  public class Point {  public int x = 0;  public int y = 0;  **//constructor**  **public Point(int a, int b){**  **x = a;**  **y = b;**  **}**  } | **b**ut it could have been written like this:  public class Point {  public int x = 0;  public int y = 0;  //constructor public  Point(int x, int y) {  this.x = x;  this.y = y;  }  } |

As we can see in the program that we have declare the name of instance variable and local variables are same. Now to avoid the confliction between them we use **this** keyword.

* From within a constructor, you can also use this keyword to call another constructor in the same class. Doing so is called an ***explicit constructor invocation***. Use this to invoke an overloaded constructor of the same class.

|  |
| --- |
| public class Rectangle {  private int x, y;  private int width, height;  public Rectangle(){  **this(0, 0, 0, 0);**  }  public Rectangle(int width, int height){  **this(0, 0, width, height);**  }  public Rectangle(int x, int y, int width, int height){  this.x = x;  this.y = y;  this.width = width;  this.height = height;  }  } |

**Note: the invocation of another constructor using this keyword must be the first line in the constructor. (e.g this (0,0,width,height) should be in the first line);**

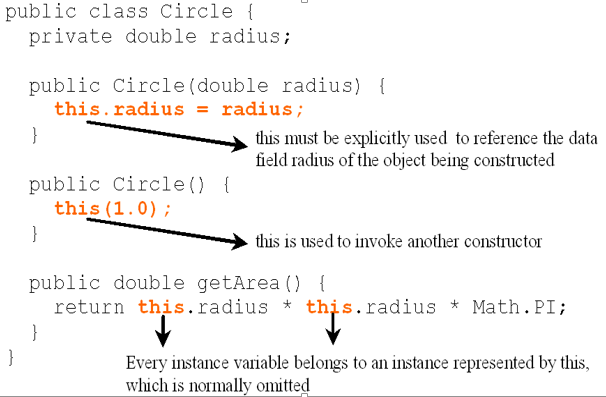


Figure 19 Example of this keyword: Circle class

* **Super Keyword**
* When extending a class you can reuse the immediate superclass constructor and overridden superclass methods by using the reserved word **super**.
* The keyword **super** refers to the superclass of the class in which **super** appears. This keyword can be used in three ways:
  + - * To call a superclass method
      * To call a superclass constructor
      * To access a hidden data fields
* Unlike data fields and methods, a superclass's constructors are not inherited in the subclass.
  + They can only be invoked from the subclasses' constructors, using the keyword **super**.
* If the keyword **super** is not explicitly used, the superclass‘s no-arg constructor is automatically invoked.
* You must use the keyword super to call the superclass constructor.
* Invoking a superclass constructor’s name in a subclass causes a syntax error.
* Java requires that the statement that uses the keyword super appear first in the constructor.

|  |  |  |
| --- | --- | --- |
| **class A{    public int a, b, c;    public  A(int p, int q, int r){      a=p;      b=q;      c=r;    }**  **public void Show(){   System.out.println("a = " + a);   System.out.println("b = " + b);   System.out.println("c = " + c);   }**  **}// end of class A public class B extends A{ public int d; public B(int l, int m, int n, int p){     super(l, m, n);     d=p; }** | **public void Show(){     super.show()     System.out.println("d = " + d); }**  **public static void main(String args[]){       B b = new B(4,3,8,7);       b.Show();     } }//end of class B**   |  | | --- | | **Output:**  **a = 4**  **b = 3**  **c = 8**  **d = 7** | |

A constructor may invoke an overloaded constructor or its superclass’s constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

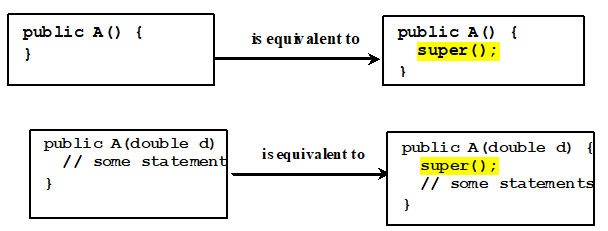


Figure 20 invocation of Super class constructor with “super”keyword

Constructing an instance of a class invokes all the superclasses’ constructors along the inheritance chain. This is called C***onstructor Chaining*.**

|  |
| --- |
| **public class Faculty extends Employee {**  **public static void main(String[] args) {**  **new Faculty();**  **}**  **public Faculty() {**  **System.out.println("(4) Faculty's no-arg constructor is invoked");**  **}**  **}**  **//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **class Employee extends Person {**  **public Employee() {**  **this("(2) Invoke Employee’s overloaded constructor");**  **System.out.println("(3) Employee's no-arg constructor is invoked");**  **}**  **public Employee(String s) {**  **System.out.println(s);**  **}**  **}**  **//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  **class Person {**  **public Person() {**  **System.out.println("(1) Person's no-arg constructor is invoked");**  **}**  } |
| **Output:**   * + 1. **Person's no-arg constructor is invoked**     2. **Invoke Employee’s overloaded constructor**     3. **Employee's no-arg constructor is invoked**     4. **Faculty's no-arg constructor is invoked** |

* You can also use super to refer to a hidden field.
* Within a class, a field that has the same name as a field in the **superclass** hides the superclass's field, even if their types are different.
  + Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through **super**.
  + Generally speaking, we don't recommend hiding fields as it makes code difficult to read.

**Note: You cannot override final methods, methods in final classes, private methods or static methods**

## **Final Classes, Abstract Classes and Interfaces**

### **Final Classes and Methods**

We may want to prevent classes from being extended. In such case, use the final modifier to indicate that a class is final and cannot be a parent class. A final class cannot be extended. This is done for reasons of security and efficiency. Accordingly, many of the Java standard library classes are final,

**Example:**

java.lang.System,  java.lang.Math and java.lang.String.

* A **final** **method** cannot be **overridden** by subclasses. This is used to prevent unexpected behavior from a subclass altering a method that may be crucial to the function or consistency of the class.
* You might wish to make a method **final** if the method has an implementation that should not be changed.
* All methods in a **final class** are implicitly final.
* A number of its methods in the Object class are final.

|  |  |
| --- | --- |
| **Example: Working with Final Class;** the first example displays the output, but the second example throws an exception because final classes cannot be inherited. | |
| **final** **class** ParentClass  {  **void** showData()  {  System.out.println("This is a method of final Parent class");  }  }  **class** MainClass  {  **public** **static** **void** main(**String** arg[])  {  ParentClass obj = new ParentClass ();  obj.showData();  }  } | **final** **class** ParentClass  {  **void** showData()  {  System.out.println("This is a method of final Parent class");  }  }  //It will throw compilation error  **class** ChildClass **extends** ParentClass  {  **void** showData()  {  System.out.println("This is a method of Child class");  }  }  **class** MainClass  {  **public** **static** **void** main(**String** arg[])  {  ParentClass obj = new ChildClass();  obj.showData();  }  } |
| **Output:** This is a method of final Parent class | **Output:** Exception in thread “main” java.lang.Error: Unresolved compilation problem: Type mismatch: cannot convert from ChildClass to ParentClass at MainClass.main(MainClass.java:21) |

### **Abstracts classes and methods**

Abstract classes are like regular classes with data and methods, but we cannot create instance of abstract classes using the new operator. An abstract class is one that cannot be instantiated. Classes that can be instantiated are known as concrete classes. If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclassed. This is typically how abstract classes come about during the design phase.

The **abstract** keyword is used to declare a class abstract. The keyword appears in the class declaration somewhere before the **class** keyword.

|  |
| --- |
| **Example: Working with Abstract Classes**  public **abstract** class Employee{  private String name, address;  private int number;  public Employee(String name, String address, int number){  System.out.println("Constructing an Employee");  this.name = name;  this.address = address;  this.number = number;  }  public double computePay(){  System.out.println("Inside Employee computePay");  return 0.0;  }  public void mailCheck(){  System.out.println("Within mailCheck of Salary class ");  System.out.println("Mailing a check to " + name + " "  + address);  }  public String toString(){  return name + " " + address + " " + number;}  public String getName(){  return name;}  public String getAddress() { return address; }  public void setAddress(String newAddress){ address =  newAddress;}  public int getNumber(){ return number;  }  }// end of class Employee |

When you would compile, using instances of Employee class, above class then you would get following error: *Employee is abstract; cannot be instantiated*

|  |
| --- |
| public class **Salary** extends **Employee** {  private double salary; //Annual salary  public Salary(String name, String address, int number, double salary){  **super(name, address, number);**  setSalary(salary);  }  public void mailCheck(){  System.out.println("Within mailCheck of Salary class ");  System.out.println("Mailing check to " + getName() + " with salary " + salary);  }  public double getSalary(){ return salary; }  public void setSalary(double newSalary){  if(newSalary >= 0.0){  salary = newSalary; }  }  public double computePay(){  System.out.println("Computing salary pay for " + getName());  return salary/52;  }  }  public class AbstractDemo{  public static void main(String [] args) {  Salary s = new Salary("Mohd ", "Ambehta, UP“, 3, 3600.00);  Salary e = new Salary("John ", "Boston, MA", 2, 2400.00);    System.out.println("Call mailCheck using Salary reference --");  s.mailCheck();  System.out.println("\n Call mailCheck using Employee reference--");  e.mailCheck();  }  } |

Here we cannot instantiate a new Employee, but if we instantiate a new Salary object, the Salary object will inherit the three fields and seven methods from Employee.

|  |
| --- |
| **This would produce following result:**  Constructing an Employee  Constructing an Employee  Call mailCheck using Salary reference --  Within mailCheck of Salary class  Mailing check to Mohd Mohtashim with salary 3600.0  Call mailCheck using Employee reference--  Within mailCheck of Salary class  Mailing check to John Adams with salary 2400. |

* **Abstract Method:** If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as **abstract**.

The **abstract** keyword is also used to declare a method as abstract. An **abstract** methods consist of a **method signature**, but no method body - its signature is followed by a semicolon, not curly braces as usual.

|  |
| --- |
| **An Abstract Class Example**: The GraphicObject class can look something like this:  abstract class GraphicObject {  int x, y;  ...  void moveTo (int newX, int newY){...}  abstract void draw();  abstract void resize(); } |

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

|  |
| --- |
| class Circle extends GraphicObject {  void draw(){... }  void resize(){...}  }  class Rectangle extends GraphicObject {  void draw(){...}  void resize(){...}  } |

**In General:**

* Abstract classes contain mixture of **non-abstract** and **abstract** methods.
* **static, private**, and **final** methods cannot be abstract, since these types of methods cannot be overridden by a subclass.
* Similarly, a **final** class cannot contain any **abstract** methods.
* Declaring a method as **abstract** has two results:
  + The class must also be declared **abstract**. If a class contains an abstract method, the class must be abstract as well.
  + When an abstract class is **subclassed**, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared **abstract**.

### **Interfaces**

Interfaces are similar to abstract classes but all methods are abstract and all data fields are static final. That is an interface is a collection of abstract methods and static final variables. An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements. Interfaces have the following properties:

* + An interface is implicitly abstract. You do not need to use the abstract keyword when declaring an interface.
  + Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
  + Methods in an interface are implicitly public.
  + Interfaces can be inherited (i.e. you can have a sub-interface). As with classes “extends” keyword is used for inheritance.
  + A class implements an interface, thereby inheriting the abstract methods of the interface.
  + Unless the class that implements the interface is abstract, all the methods of the interface need to be defined and implemented in the class.
  + A concrete class may implement one or several interfaces, supplying the code for all the methods.
* An interface is similar to a class in the following ways:
  + An interface can contain any number of methods.
  + An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
  + The bytecode of an interface appears in a .class file.
  + Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.
* However, an interface is different from a class in several ways, including:
  + You cannot instantiate an interface.
  + An interface does not contain any constructors.
  + All of the methods in an interface are abstract.
  + An interface cannot contain instance fields.
  + An interface is not extended by a class; it is implemented by a class.
  + An interface can extend multiple interfaces.
* The **interface** keyword is used to declare an interface.

|  |
| --- |
| public **interface** NameOfInterface{  //Any number of final, static fields  //Any number of abstract method declarations\  } |

A class uses the **implements** keyword to implement an interface. The **implements** keyword appears in the class declaration following the extends portion of the declaration.

|  |  |
| --- | --- |
| **Example: Working with interface**  **interface Animal** {  public void eat();  public void travel();  }  **public class Mammal implements Animal**{  public void eat(){  System.out.println("Mammal eats");  }  public void travel(){  System.out.println("Mammal travels"); }  public int noOfLegs(){  return 0; }  public static void main(String args[]){  Mammal m = new Mammal();  m.eat();  m.travel();  }  } | Output:  Mammal eats  Mammal travels |

An interface can **extend** another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

|  |
| --- |
| **public interface Sports**{  public void setHomeTeam(String name);  public void setVisitingTeam(String name);  }  **public interface Football extends Sports**{  public void homeTeamScored(int points);  public void visitingTeamScored(int points);  public void endOfQuarter(int quarter);  }  **public interface Hockey extends Sports**{  public void homeGoalScored();  public void visitingGoalScored();  public void endOfPeriod(int period);  public void overtimePeriod(int ot);  } |

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

**Note that: an interface can extend more than one parent interface**

|  |
| --- |
| **public interface Hockey extends Sports, Event** |

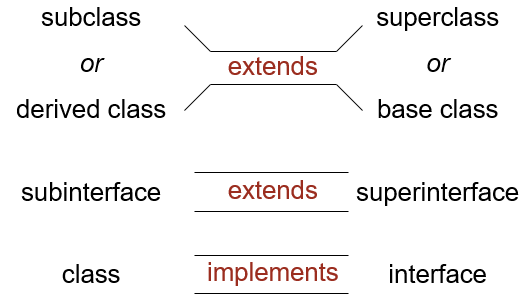


Figure 21 inheritance relations class with class, class with interface and interface with interface

**In General:**

* In Java, a subclass can extend only one superclass.
* In Java, a subinterface can extend several superinterface
* In Java, a class can implement several interfaces — this is Java’s form of ***multiple inheritance***.
* An abstract class can have code for some of its methods; other methods are declared **abstract** and left with no code.
* An interface only lists methods but does not have any code.
* Inheritance plays a dual role:
* A subclass reuses the code from the superclass.
* A subclass (or a class that implements an interface) inherits the data type of the superclass (or the interface) as its own secondary type.
* Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods.
* If an abstract class contains *only* abstract method declarations, it should be declared as an interface instead.
* Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way.
* By comparison, abstract classes are most commonly subclassed to share pieces of implementation.
* A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

## **Package**

A Package (named collections of classes) can be defined as a grouping of related types(classes and interfaces) providing access protection and name space management. Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier etc.

Java has numerous built in packages: **java.io, java.lang, java.util**

Programmers can define their own packages to bundle group of classes/interfaces etc. It is a good practice to group related classes implemented by you so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes and interfaces types that you want to include in the package.

* The **package** statement should be the first line in the source file.
* There can be only **one package** statement in **each source file**, and it applies to all types in the file.
* It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.
* If a class wants to use another class in the same package, the package name does not need to be used.
* Classes in the same package find each other without any special syntax.
* A class file can contain any number of **import** statements.
* The import statements must appear after the package statement and before the class declaration.

Let us look at an example that creates a package called **animals**.

|  |
| --- |
| **Example: collecting a number of classes in a package**  package animals;  interface Animal {  public void eat();  public void travel();  }  //--------------------------------------------------------------------------  package animals;  public class MammalInt implements Animal{    public void eat(){ System.out.println("Mammal eats"); }  public void travel(){ System.out.println("Mammal travels"); }    public int noOfLegs(){ return 0; }  public static void main(String args[]){  MammalInt m = new MammalInt();  m.eat();  m.travel();  }  } |

* The fully qualified name of the class can be used.

***For example***: *animals.MammalInt*

* The package can be imported using the import keyword and the wild card (\*).

***For example***: *import animals.\*;*

* The class itself can be imported using the import keyword.

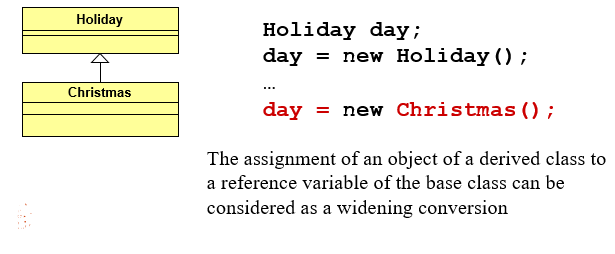
***For example****: import animals.MammalInt;*

# Chapter 4: Polymorphism

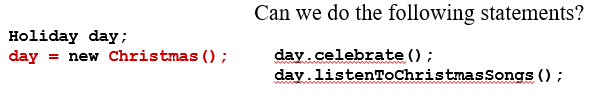
The inheritance relationship enables a subclass to inherit features from its superclass with additional new features. A subclass is a specialization of its superclass. Every instance of a subclass is an instance of its superclass, but not vice versa. A subclass possesses all the attributes and operations of its superclass (because a subclass inherited all attributes and operations from its superclass). This means that a subclass object can do whatever its superclass can do.

Polymorphism (from the Greek, meaning “many forms”) is the ability of an object to take on many forms. All interaction with an object occurs through object reference variables. An object reference variable holds the reference (address, the location) of an object. An object reference can refer to an object of its class, or to an object of any class derived from it by inheritanc**e.**

**For example**, if the **Holiday** class is used to derive a child class called **Christmas**, then a **Holiday** reference could actually be used to point to a **Christmas** object.



|  |
| --- |
| **class Holiday {  public void celebrate()   {…} }**  **//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  class Christmas extends Holiday {  public void celebrate()   {…}  public void listenToChristmasSongs()   {…}  } |



The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed. The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object. A reference variable can refer to any object of its declared type or any subtype of its declared type.

|  |
| --- |
| **public class Shape**{  public double computeArea(){ return -1; }  public double computeCircumfrence(){ return -1; }  }  **public class Circle extends Shape**{  private double radius;  public Circle(double r){ this.radius = r; }  public double computeArea(){ return Math.PI\*this.radius \* this.radius; }  public double computeCircumfrence(){ return 2\*Math.PI\*this.radius; }  }  **public class Rectangle extends Shape**{  private double width;  private double height;  public Rectangle(double w, double h){  this.width = 0;  this.height = 0;  }  public double computeArea(){ return this.width \* this.height; }  public double computeCircum(){ return 2\*(this.width \*this.height); }  }  **public class ShapeTest**{  public static void main(String args[]){  Shape[] shapes = new Shape[3];  shapes[0] = new Circle(4.0);  shapes[1] = new Rectangle(5, 3);  shapes[2] = new Circle(1.8);  double totalArea = 0;  for(int i = 0;i<shapes.length;i++){ totalArea+=shapes[i].computeArea(); }  System.out.println(“Total area = “ +totalArea);  }} |

**Out Put: Total area = 60.44424265506762**

A reference variable of type Shape is used to refer objects of types of its subclasses (Circle and rectangle) What is the output of the above code if you comment out **computeArea**() method in Circle and Rectangle classes? Out Put: Total area = -3.0

When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called. (See the example on shapes). Suppose an object o is an instance of class c1,c2, ….cn-1 and cn, where c1 is subclass of c2, c2 is subclass of c3,….and cn-1 is a subclass of cn. That is cn is the most general class and c1 is the most specific class. In java cn is the Object class. If o invokes a method m, the JVM searches the implementation for m in c1,c2,…cn-1 and cn in this order, until it is found. The output illustrates that the appropriate methods for each class are indeed invoked based on the type of the object to which shapes refers. This is called dynamic binding or late binding. However, if a program needs to perform a subclass-specific operation on a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as downcasting. This enables the program to invoke subclass methods that are not in the superclass.

* **Upcasting & Downcasting:** Substituting a subclass instance for its superclass is called "*upcasting". Upcasting is always safe because a subclass* instance possesses all the properties of its superclass and can do whatever its superclass can do. You can revert a substituted reference back to the original subclass. This is called "*downcasting".*

|  |
| --- |
| ***For example,***  Shape shape = new Circle(5.0);// upcast is safe  Circle aCircle = (Circle)shape;  //downcast needs the casting operator |

For Example if a Rectangle class had getDiagonal() method and if we want to use this method we must first downcast it.

|  |
| --- |
| if(shap[1] instanceOf Rectangle)  Rectangle r = (Rectangle)shapes[1]; |

We cannot treat a superclass object as a subclass object because a superclass object is not an object of any of its subclasses. Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error. Downcasting requires *explicit type casting operator in the form of prefix operator (new‐type). Downcasting is not always* safe, and throws a runtime ClassCastException if the instance to be downcasted does not belong to the correct subclass. A subclass object can be substituted for its superclass, but the reverse is not always true. It is possible to display each shape’s type as a string.

|  |  |
| --- | --- |
| for(int i = 0;i<shapes.length;i++){  System.out.printf( “Shape %d is a %s\n", i,  shapes[ i].getClass().getName());  } | The output is  Shape 0 is a Circle  Shape 1 is a Rectangle  Shape 2 is a Circle |

## **Polymorphism with Overloading and Overriding**

**Polymorphism** can also be achieved through method overriding and method overloading.

* + - **Overridden methods:** Are methods that are redefined within an inherited or subclass. They have the same signature and the subclass definition is used. The implementation in the subclass overrides (replaces) the implementation in the superclass. In other words an instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass ***overrides*** the superclass's method.

|  |
| --- |
| **Example:** Overriding  class Rectangle {  private int x, y, w, h;  public String toString() {  return “x = “ + x + “, y = “ + y +“, w = “ + w + “, h = “ + h;  }  }  class DecoratedRectangle extends Rectangle {  private int borderWidth;  public String toString() {  return super.toString() + “, borderWidth = “ + borderWidth;  }  } |

**Method Overriding rules:**

* + The argument list should be exactly the same as that of the overridden method.
  + The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
  + The access level cannot be more restrictive than the overridden method's access level. However the access level can be **less restrictive** than the overridden method's access level.
  + Instance methods can be overridden only if they are inherited by the subclass.
  + Constructors cannot be overridden.
  + A method declared **final** cannot be overridden.
  + A method declared **static** cannot be overridden but can be re-declared.
  + If a method cannot be inherited then it cannot be overridden.
  + A subclass within the same package as superclass can override any superclass method that is not declared **private** or **final**.
  + A subclass in a different package can only override the non-final methods declared public or protected.

**For example:** if the super class method is declared **public** then the overriding method in the subclass cannot be either **private** or **protected**.

* + - **Method Overloading: *Overloaded methods*** are methods with the same *name signature* but either a different number of parameters or different types in the parameter list. Compiler automatically select the most appropriate method based on the parameter supplied.

|  |
| --- |
| Example  public class MultiplyNumbers {   public int Multiply(int a, int b) { return a \* b; }   public int Multiply(int a, int b, int c) { return a\*b\*c; }  } |

# **Chapter 5: Exception Handling**

## **Exception handling overview**

An exception is a problem that arises during the execution of a program. It is a representation of an error condition or a situation that is not the expected result of a method/program.

### **The causes of exceptions**

An exception can occur for various reasons:

* Attempting to divide by zero (arithmetic exception)
* Reading a decimal value when an integer is expected (number format exception)
* Attempting to write to a file that doesn't exist (I/O exception).
* Access an element past the end of an array. (ArrayIndexOutOfBoundsException)
* or referring to a nonexistent character in a string (StringIndexOutOfBounds exception).
* Invoke a method on a reference variable with null value. (NullPionterException)
* A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner. No matter how well-designed a program is, there is always the chance that some kind of error will arise during its execution. A program that does not provide code for catching and handling exceptions will terminate abnormally, and may cause serious problems.

A well-designed program should include code to guard against errors and other exceptional conditions when they arise. This code should be incorporated into the program from the very first stages of its development. That way it can help identify problems during development. In Java, the preferred way of handling such conditions is to use **exception handling -** a divide-and-conquer approach that separates a program's normal code from its error-handling code.

### **The Throwable class hierarchy**

The Java class library contains a number of **predefined** exceptions. All exception classes are subtypes of the **java.lang.Exception** class. The exception class is a subclass of the **Throwable** class. Other than the exception class there is another subclass called **Error** which is derived from the **Throwable** class. The **Throwable** class is contained in the **java.lang** package and subclass of Throwable are contained in various packages.

**Example:** Errors related to GUI components are included in the **java.awt.**

Errors are not normally trapped from the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. **Errors** are generated to indicate errors generated by the runtime environment.

Example: JVM is out of Memory. Normally programs cannot recover from errors.

Figure 22 Throwable Hierarchy

Exception handling works in Java in the three categories of exceptions:

1. **Checked exceptions:** A checked exception is one that can be analyzed (can’t be ignored) by the Java compiler. That is when the compiler encounters one of these exceptions it checks whether the program either handles or declares the exception. A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs.
2. **Runtime exceptions:** Runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation. ***Runtime Exception*** is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors
3. **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. **For example**, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation. **Errors** are thrown by JVM and represented in the ***Error*** class. The ***Error*** class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

***Runtime-Exception***, ***Error*** and their subclasses are known as ***unchecked* *exceptions***. All other exceptions are known as ***checked exceptions***, meaning that the compiler forces the programmer to check and deal with the exceptions.

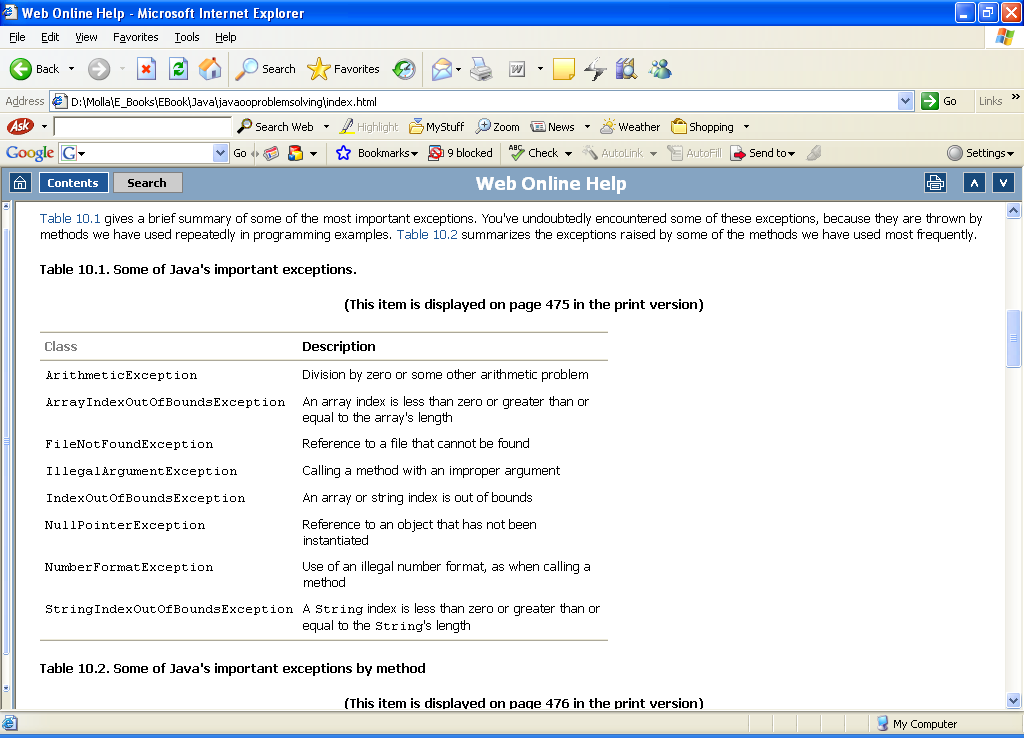
### **Handling of an exception**

**Exception handling** is the technique of catching the exceptions that might be thrown sometime in the future during runtime. Exceptions can be handled in traditional way of handling errors within a program with Java's default exception-handling mechanism or using Exception class defined in Java API. **Traditional way of Handling Errors:** has several problems. Programmer must remember to always check the return value and take appropriate action. This requires much code (methods are harder to read) and something may get overlooked.

Consider the following example:

|  |  |
| --- | --- |
| **Example:** Dividing a number to Zero | **Solution:** the error handling code is built right into the algorithm. |
| public double avgFirstN(int N) {  int sum = 0;  for(int k = 1; k <= N; k++)  sum += k;  return sum/N; // What if N is 0?  } | public double avgFirstN(int N) {  int sum = 0;  if (N <= 0) {  System.out.println("ERROR avgFirstN: N <= 0. Program terminating.");  System.exit(0);  }  for (int k = 1; k <= N; k++)  sum += k;  return sum/N; // What if N is 0?  } // avgFirstN( |

**Common Exceptions include**:



#### **Exception handle Methods.**

Java exception handling is a mechanism for handling exception by detecting and responding to exceptions in a systematic, uniform and reliable manner. Exception handling in Java is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. It is accomplished through the “**try – catch**” mechanism, or by a “**throws or throw**” clause in the method declaration.

1. **Try-Catch Mechanism:** Wherever your code may trigger an exception, the normal code logic is placed inside a block of code starting with the “try” keyword: After the try block, the code to handle the exception should it arise is placed in a block of code starting with the “catch” keyword. You might like to think of these as "try to do this task" and if there is an exception "catch the exception and do something about it". A try/catch block is placed around the code that might generate an exception.

Code within a **try/catch** block is referred to as protected code. You may also write an optional “**finally**” block. This block contains code that is ALWAYS executed, either after the “try” block code, or after the “catch” block code. **Finally** blocks can be used for operations that must happen no matter what (i.e. cleanup operations such as closing a file) generally, the try statement contains and guards a block of statements.

|  |
| --- |
| Syntax of **try catch** |
| **try{**  codes that may throw exception(s)  **}**  **catch (exception\_type1 identifier){**  //how do you want to deal with this exception  **}**  **catch (exception\_type2 identifier){**  //how do you want to deal with this exception  **}**  // you can use multiple catches to handle different exceptions |

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter. Only one catch block, that is the first applicable one, will be executed. If no exceptions arise during the execution of the try block, the catch blocks are skipped.

Example:



A catch block will catch exceptions of the class specified, including any exceptions that are subclasses of the one specified. The order in which exceptions are specified in catch blocks is important. A catch block for a subclass type appears before a catch block for a superclass or A more specific catch block must precede a more general one in the source. Failure to meet this ordering requirement causes a compiler error.

|  |  |
| --- | --- |
| **Example: Ordering of Exception** | |
| **try {**  **..…..**  **}**  **catch(Exception ex){**  **…….**  **}**  **catch (RuntimeException ex) {**  **……..**  **}**  (a) Wrong order | **try {**  **..…..**  **}**  **catch(RuntimeException ex){**  **……**  **}**  **catch (Exception ex ) {**  **……..**  **}**  **(b) Correct Order** |

* **The finally Keyword:** The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred. Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of the catch blocks and has the following syntax:

|  |
| --- |
| **Syntax**: Using finally keyword in try – catch exception handling |
| **try{**  codes that may throw exception(s)  **}**  **catch (exception\_type1 identifier){**  //how do you want to deal with this exception  **}**  **catch (exception\_type2 identifier){**  //how do you want to deal with this exception  **}**  // you can use multiple catches to handle different exceptions  **finally {**  // code that must be executed under successful or unsuccessful conditions  **}** |

The finally block always executed regardless an exception or not except the following conditions:

* + The death of the thread
  + The use of System.exit( )
  + Turning off the power to the CPU
  + An exception arising in the finally block itself

|  |  |
| --- | --- |
| **Example:** | **Output:** |
| public class ExcepTest{  public static void main(String args[]){  int a[] = new int[2];  **try{**  System.out.println("Access element three :" + a[3]);  **}**  **catch(ArrayIndexOutOfBoundsException e){**  **System.out.println("Exception thrown :" + e);**  **}**  **finally{**  **a[0] = 6;**  **System.out.println("First element value: " +a[0]);**  **System.out.println("The finally statement is executed");**  **}**  }  } | **Exception thrown:java.lang.ArrayIndexOutOfBoundsException: 3**  **First element value: 6**  **The finally statement is executed** |

**Note the followings**:

* A catch clause cannot exist without a try statement.
* It is not compulsory to have finally clauses whenever a try/catch block is present.
* The try block cannot be present without either catch clause or finally clause.
* Any code cannot be present in between the try, catch, finally blocks.
  + - **The throws/throw Keywords:** In any method that might throw an exception, you may declare the method as “**throws**” that exception, and thus avoid handling the exception yourself. The **throws** keyword appears at the end of a method's signature.

|  |
| --- |
| **Example**  public void myMethod throws IOException {  normal code with some I/O } |

Every method must declare the types of **checked exceptions** it might throw using **throws** keyword. A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas.

|  |
| --- |
| public void myMethod() **throws Exception1, Exception2, …. ExceptionN** |

When the program detects an error, the program can create an instance of an appropriate exception type and throw it by using the **throw** keyword.

|  |  |
| --- | --- |
| **Declaration Example:**  **throw** new TheException();  **or**  TheException ex = new TheException(); **throw** ex; | **Example;**  public void openFile(String fileName) **throws IOException**{  open the file using the input file name;  if (can not open file) **throw new IOException();**  read the file,  } |

In general, each exception class in the java API has al least two **constructor**: a no-arg constructor, and a constructor with a String arguments that describes the exception. This argument is called the exception message, which can be obtained using **getMessage().**

|  |
| --- |
| public void setRadius(double newRadius)  **throws IllegalArgumentException** {  if (newRadius >= 0)  radius = newRadius;  else  **throw new IllegalArgumentException( "Radius cannot be negative");**  } |

## **User defined exceptions**

It is also possible to create user defined exceptions in Java. Keep the following points in mind when writing user defined exception classes:

* All exceptions must be a child of **Throwable**.
* If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the **Exception** class.
* If you want to write a runtime exception, you need to extend the **RuntimeException** class.
* You can define our own Exception class as below:

|  |
| --- |
| **Syntax:**  class MyException **extends Exception{ …}** |

You just need to extend the Exception class to create your own Exception class. These are considered to be **checked exceptions**.

The following **InsufficientFundsException** class is a user-defined exception that extends the **Exception** class, making it a checked exception.

|  |  |
| --- | --- |
| **Example**:  import java.io.\*;  public class CheckingAccount{  private double balance;  private int number;  public CheckingAccount(int number) { this.number = number; }  public void deposit(double amount) { balance += amount; }  public double getBalance() { return balance; }  public int getNumber() { return number; }  public void withdraw(double amount) throws InsufficientFundsException {  if(amount <= balance) {  balance-= amount;  }  else{  double needs = amount - balance;  throw new InsufficientFundsException(needs);  }  }  }  public class BankDemo{  public static void main(String [] args) {  CheckingAccount c = new CheckingAccount(101);  System.out.println("Depositing $500...");  c.deposit(500.00);  try {  System.out.println("\nWithdrawing $100...");  c.withdraw(100.00);  System.out.println("\nWithdrawing $600...");  c.withdraw(600.00);  }catch(InsufficientFundsException e) {  System.out.println("Sorry, but you are short $“ + e.getAmount());  e.printStackTrace();  **}**  **}** | **// File Name InsufficientFundsException.java**  import java.io.\*;  public class InsufficientFundsException extends Exception{  private double amount;  public InsufficientFundsException(double amount){  this.amount = amount;  }  public double getAmount(){ return amount; }  }  **Output:**  **Depositing $500...**  **Withdrawing $100...**  **Withdrawing $600...**  **Sorry, but you are short $200.0**  **InsufficientFundsException**  **at CheckingAccount.withdraw(CheckingAccount.java:25)**  **at BankDemo.main(BankDemo.java:13)** |

# **Chapter 6: Files and Streams**

## **Introduction**

Whenever our programs have produced output, it may be sent to either the Java console, a text area, or some other GUI component. These destinations are transitory, in the sense that they reside in the computer's primary memory and exist only so long as the program is running. Or it may be sent to relatively permanent storage medium. A **file** is a collection of data stored on a disk or on some other relatively permanent storage medium. A file's existence **does not depend** on a running program.

I/O = Input/Output

In this context it is input to and output from programs. **Input** refers to information or data read from some external source into a running program. **Output** refers to information or data written from a running program to some external destination. Input can be from keyboard, a file, or network connection etc. Output can be to display (screen) or a file. Advantages of file I/O

* + Permanent copy
  + Output from one program can be input to another
  + Input can be automated (rather than entered manually)

## **STREAM**

Stream an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.) It acts as a buffer between the data source and destination. A stream connects a program to an I/O object.

* **Input stream**: a stream that provides input to a program

System.in is a Standard input stream, it connects a program to the keyboard

* **Output stream**: a stream that accepts output from a program

System.out is a Standard output stream, it connects a program to the screen

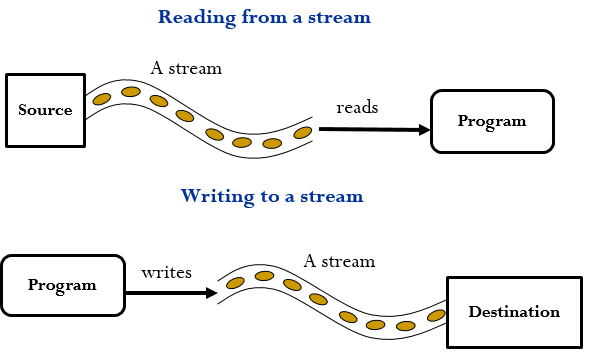


Figure 23 Reading and Writing to a Stream

## **Types of Stream**

Java defines two types of streams:  **byte based** and **character based**.

* 1. **Byte Based:** Streams provide convenient way for handling input and output of **8-bit** **bytes** and are used for input/output of binary data. Files created by byte based streams are called **binary files**. These files are easily read by the computer but not humans. Binary data are not very portable (platform dependent). On some systems an integer might be 16 bits, and on others it might be 32 bits, so even if you know that a Macintosh binary file contains integers, that still won't make it readable by Windows/Intel programs.
  2. **Character** **Based**: Streams provide a convenient way for handling input and output of characters (**text files**). Files created using character based streams are called **text file.** A **text file** is processed as a sequence of characters. A text file created by a program on a Windows/Intel computer can be read by a Macintosh program. Text files are portable.

**Note that**: Text files are human readable files. They are universal and can be edited with many different programs such as NOTEPAD.

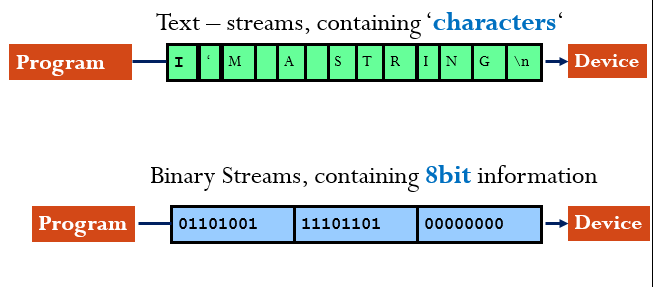


Figure 24 Types of Stream

Data stored in a text file are represented in human-readable form. Data stored in a binary file are represented in binary form. You cannot read binary files. Binary files are designed to be read by programs. For example, the Java source programs are stored in text files and can be read by a text editor, but the Java classes are stored in binary files and are read by the JVM. The advantage of binary files is that they are more efficient to process than text files.

The Java I/O package gives classes support for reading and writing data to and from different input and output sources including Arrays, files, strings, sockets, memory and other data sources. The **java.io** package provides more than 60 input/output classes (stream). These classes are used to manipulate **binary** and **text** files. Here, we will learn how to create files and how to perform input and output operations on their data using the Java classes designed specifically for this purpose.

## **I/O classes**

Streams in JAVA are Objects. Having

* + 2 types of streams (text / binary) and
  + 2 directions (input / output)

Results in 4 base-classes dealing with I/O:

* **InputStream:** byte-input
* **OutputStream:** byte-output
* **Reader:** text-input
* **Writer:** text-output

Generally speaking, **Binary files** are processed by ***subclasses*** *of* **InputStream** *and* **OutputStream**; and **Text files** are processed by ***subclasses*** of **Reader** and **Writer.** Both of which are **streams** despite their names. Base classes are used to describe the basic functionality required (InputStream, OutStream, Reader and Writer) and derived classes provided the functionality for specific kinds of I/O environments (Files, Pipes, Networks, IPC, etc.)

### **Byte Based Stream**

Byte based streams classes are defined by two class hierarchies. At the top are two **abstract classes**:

* + - **InputStream**
    - **OutputStream**

Both of these classes are **abstract** and have several **concrete subclasses** that handle the difference between various sources, such as disk, files or sockets. The abstract classes **InputStream** and **OutputStream** declare several **abstract methods** that all subclasses implement. Two of the most important methods are **read( )** and **write( )**, which respectively read and write single byte.

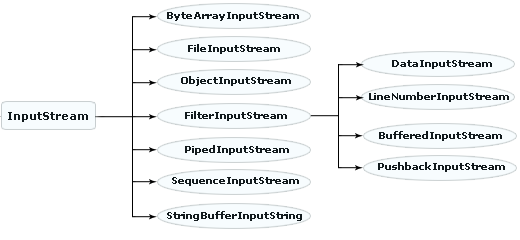


Figure 25 Byte based Input Stream

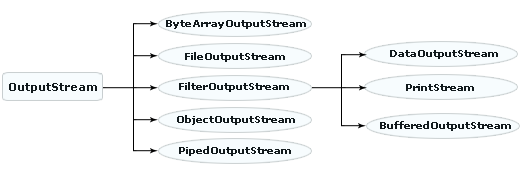


Figure 26 Byte based Output Stream

#### **Reading and Writing to a Binary file**

* **Writing data to a file requires** **three steps**:
  1. Connect an output stream to the file. The output stream serves as a channel between the program and a named file. The output stream opens the file and gets it ready to accept data from the program. If the file already exists, then opening the file will destroy any data it previously contained.

If the file doesn't yet exist, then it will be created from scratch.

* 1. Write text data into the stream, possibly using a loop. Once the file is open, the next step is to write the text to the stream, which passes the text on to the file. This step may require a loop that outputs one line of data on each iteration.
  2. Close the stream: Finally, once all the data have been written to the file, the stream should be closed. This also has the effect of closing the file. Even though Java will close any open files and streams when a program terminates normally, it is good programming practice to close the file yourself with a close () statement. This reduces the chances of damaging the file if the program terminates abnormally.
* **Reading data from a file: require three steps**

1. Connect an input stream to the file. The Input stream opens the file and gets it ready to be read by the program. If the file doesn’t exist, a compilation error will happen. And an exception should have to be thrown.
2. Read the text data using a loop.
3. Close the stream.

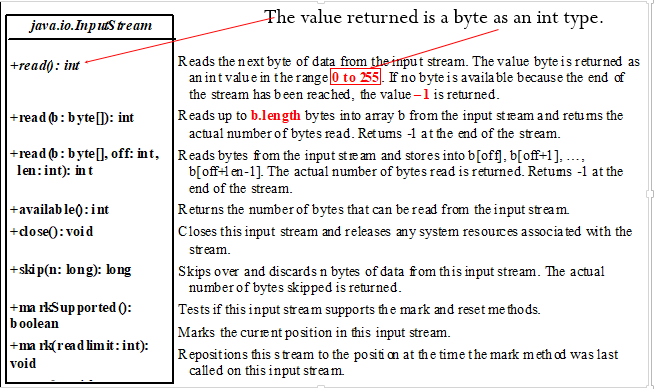


Figure 27 Methods of InputStream class

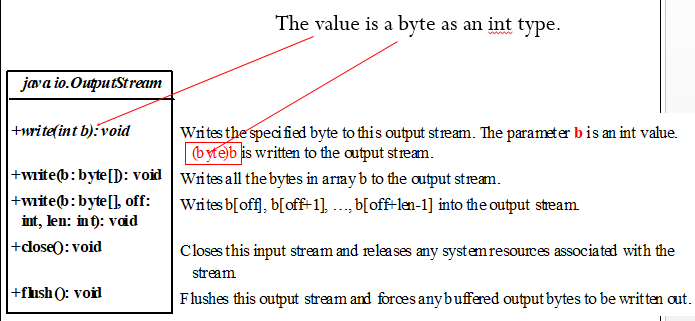


Figure 28 Methods of OutputStream class

#### **Subclasses of byte based Streams**

* + 1. **FileInputStream/FileOutputStream:** associates a binary input/output stream with an external file. All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses. FileInputStream/FileOuptputStream is for reading/writing bytes from/to files. All the methods in FileInputStream/FileOuptputStream are inherited from its superclasses. FileInputStream/FileOuptputStream does not introduce new methods.
* **FileInputStream**: To construct a FileInputStream, use the following constructors:

|  |
| --- |
| public FileInputStream(String filename) //Creates a FileInputStream from a file name.  public FileInputStream(File file) //Creates a FileInputStream from a File object. |

A **java.io.FileNotFoundException** would occur if you attempt to create a FileInputStream with a nonexistent file.

* **FileOutputStream:** To construct a FileOutputStream, use the following constructors:

|  |
| --- |
| public FileOutputStream(String filename) // Create a FileOutputStream from a file name.  public FileOutputStream(File file) //Create a FileOutputStream from a File object.  public FileOutputStream(String filename, boolean append) // If append is true, data is appended to the existing file.  public FileOutputStream(File file, boolean append) // If append is true, data is appended to the existing file. |

If the file does not exist, a new file would be created. If the file already exists, the first two constructors would delete the current contents in the file. To retain the current content and append new data into the file, use the last two constructors by passing true to the append parameter.

|  |
| --- |
| Example: Working with FileInputStream and FileOutputStream  import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  public class CopyBytes {  public static void main(String[] args) throws IOException {  FileInputStream in = new FileInputStream("SourceFile.txt"); ;  FileOutputStream out = new FileOutputStream("TargetFile.txt“);  int c;  while ((c = in.read()) != -1) {  out.write(c);  }  if (in != null) { in.close(); }  if (out != null) { out.close(); }  }} |

* + 1. **FilterInputStream/FilterOutputStream:** *Filter streams* are streams that filter bytes for some purpose. The basic byte input stream provides a read method that can only be used for reading bytes. Using a filter class enables you to read integers, doubles, and strings instead of bytes and characters. **FilterInputStream** and **FilterOutputStream** are the base classes for filtering data (e.g. DataInputStream and DataOutputStream, BufferedInputStream and BufferedOutputStream…).
* **DataInputStream\DataOutputStream**: When you need to process primitive numeric types, use **DataInputStream** and **DataOutputStream** to filter bytes. **DataInputStream** reads bytes from the stream and converts them into appropriate primitive type values or strings. **DataOutputStream** converts primitive type values or strings into bytes and output the bytes to the stream. **DataInputStream** extends **FilterInputStream** and implements the **DataInput** interface.

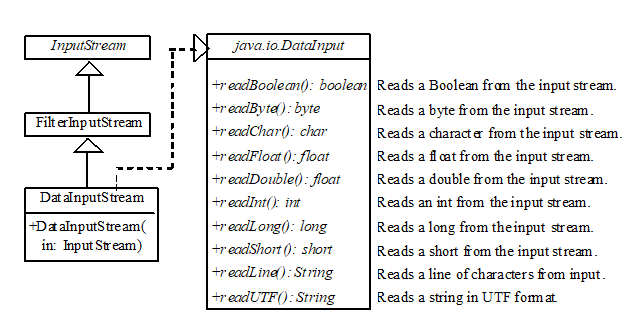


Figure 29 DataInputStream methods

**DataOutputStream** extends **FilterOutputStream** and implements the **DataOutput** interface.

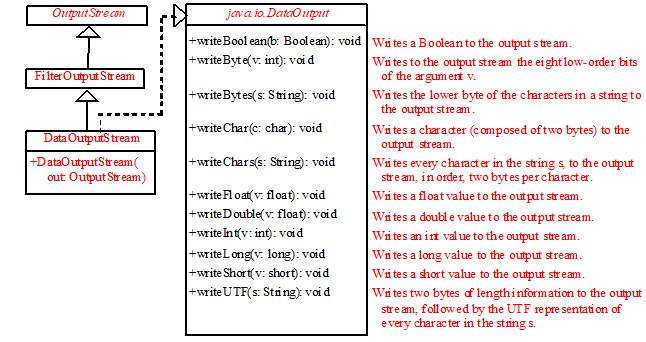


Figure 30 DataOutputStream methods

Data streams are used as wrappers on existing input and output streams to filter data in the original stream. They are created using the following constructors:

|  |
| --- |
| public DataInputStream(InputStream instream)  public DataOutputStream(OutputStream outstream) |

The statements given below create data streams. The first statement creates an input stream for file in.dat; the second statement creates an output stream for file out.dat.

|  |
| --- |
| DataInputStream infile = new DataInputStream(new FileInputStream("in.txt"));  DataOutputStream outfile = new DataOutputStream(new FileOutputStream("out.txt")); |

|  |
| --- |
| **Example**: Working with DataInputStream and DataOutputStream  import java.io.\*;  public class TestDataStream {  public static void main(String[] args) throws IOException {  // Create an output stream for file temp.txt  DataOutputStream output = new DataOutputStream(new FileOutputStream("temp.txt“));  // Write student test scores to the file  output.writeUTF("John");  output.writeDouble(85.5);  output.writeUTF("Jim");  output.writeDouble(185.5);  output.writeUTF("George");  output.writeDouble(105.25);  output.close(); // Close output stream  //--------------------------------------------------  // Create an input stream for file temp.txt  DataInputStream input = new DataInputStream(new FileInputStream("temp.txt”));  // Read student test scores from the file  System.out.println(input.readUTF() + " " +input.readDouble());  System.out.println(input.readUTF() + " " +input.readDouble());  System.out.println(input.readUTF() + " " +input.readDouble());  }  } |

* **BufferedInputStream and BufferedOutputStream**: are another subclass of FilterInputStream and FilterOutputStream. Most of the examples we've seen so far use ***unbuffered* I/O**. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive. To reduce this kind of overhead, the Java platform implements ***buffered* I/O streams**. Buffered input streams read data from a memory area known as a *buffer*; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full. BufferedInputStream/BufferedOutputStream Can be used to speed up input and output by reducing the number of reads and writes. BufferedInputStream/BufferedOutputStream does not contain new methods. All the methods of BufferedInputStream/BufferedOutputStream are inherited from the InputStream/OutputStream classes. To create BufferedInputStream and BufferedOutputStream the following Constructors are used.

|  |
| --- |
| // Create a BufferedInputStream  public BufferedInputStream(InputStream in)  public BufferedInputStream(InputStream in, int bufferSize)    // Create a BufferedOutputStream  public BufferedOutputStream(OutputStream out)  public BufferedOutputStream(OutputStreamr out, int bufferSize) |

### **Character based Streams**

Character based streams are also defined by two class hierarchies. At the top are two abstract classes:

* + Reader
  + Writer

These abstract classes handle Unicode characters. These abstract classes Reader and Writer declare many abstract methods which are implemented by other subclasses. Two most important methods are read( ) and write( ), which respectively read and write one character. Both methods are declared as abstract inside Reader and Writer. All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWirter. The CopyCharacters example illustrates these classes.

Constructors of FileWriter stream

|  |
| --- |
| FileWriter(File file) // Constructs a FileWriter object given a File object.  FileWriter (File file, boolean append)//Constructs a FileWriter object given a File object.  FileWriter(String fileName) // Constructs a FileWriter object given a file name.  FileWriter(String fileName, boolean append) |

 Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.

* Constructors of FileReader stream

|  |
| --- |
| **FileReader** (File file) // Constructs a FileReader object given a File object.  **FileReader** (String fileName)  //Constructs a FileReader object given a file name. |

|  |
| --- |
| Example: Working with FileWriter and FileReader  import java.io.\*;  public class CopyCharacter {  public static void main(String[] args) throws IOException  {  FileReader inputStream =new FileReader(“File.txt");  FileWriter outputStream = new FileWriter(“File1.txt");  int c;  while ((c = inputStream.read())!= -1) {  outputStream.write(c);  }  if (inputStream != null) { inputStream.close(); }  if (outputStream != null) { outputStream.close(); }  }  } |

Other classes which could be used for writing to a text file are: all classes are subclasses of Writer class:

BufferedWritter, CharArrayWritter, PipedWritter, StringWritter, PrintWritter.

Each of these classes have their own constructors and methods. Some of the classes which are used for reading text files are: all classes are subclasses of Reader class:

BufferedReader, LineNumberReader, CharArrayReader, InputStreamReader, FileReader, FilterReader, PushbackReader, PipedReader, StringReader

* **Line-Oriented I/O**

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes **BufferedReader** and **PrinterWriter.** The **CopyLines** example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time. Joining a BufferedReader and a FileReader.

* BufferedReader
  + Has readLine() method
  + But lacks a constructor that can take file name
* FileReader
  + Has constructor which can take file name
  + But lacks readLine() method
* Combine them together as follows:

BufferedReader inStream = new BufferedReader(new FileReader(fileName));

Here the **BufferedReader** will read from a file. Now it is possible to use: inStream.readLine() to read one line at a time from the file. An important fact about **readLine**() is that it will return null as its value when it reaches the end of the file. That is readLine() does not return the end-of-line character as part of the text it returns.

**PrintWriter:-**  object takes Strings and data types from program and write in output stream. We can use the print( ) and println( ) methods just like System.out.println( ) to write any type of data to the console.

|  |
| --- |
| **String s = “Hello World”;**  **p.println( s );**  **int i = 55;**  **p.print( i );** |

For real world programs the recommended way of writing to the console is **PrintWriter** class.

To create the object we use the following constructor.

|  |
| --- |
| **PrintWriter p = new PrintWriter(System.out);** |

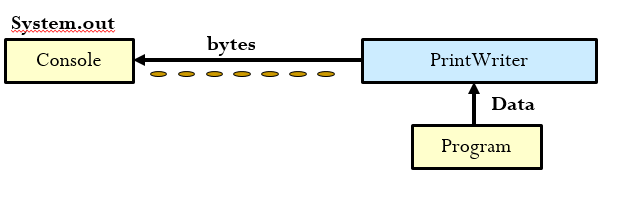


Figure 31 PrinterWriter

|  |
| --- |
| Example: Reading and Writing one line at a time  import java.io.\*;  public class CopyLines {  public static void main(String[] args) throws IOException {  BufferedReader inputStream = new BufferedReader(new FileReader(“File.txt"));  PrintWriter outputStream = new PrintWriter(new FileWriter(“File1.txt"));  String l;  while ((l = inputStream.readLine()) != null) {  outputStream.println(l);  }  if (inputStream != null) { inputStream.close(); }  if (outputStream != null) { outputStream.close(); }  }  } |

## **The File Class**

Before we see how to read and write binary and text files let’s see File class. The File class does not permit any I/O, instead it provides a means of querying/modifying filename or pathnames (the class would be better termed FileName). java.io.File is the central class in working with files and directories. Files and directories are both represented by File objects. When a File object is created, the system doesn't test to see if a corresponding file/directory actually exists; you must call exists () to check.

Constructors

|  |
| --- |
| File *f* = new File (String *path*); //Create File object for default directory (usually where program is located).  File *f* = new File (String *dirpath*, String *fname*); //Create File object for directory path given as string.  File *f* = new File (File *dir*, String *fname*); //Create File object for directory. |

public static constants-> String *s* = File.separator; (Default path separator (eg, "/" in Unix, "\" in Windows).

* *Getting Attributes* ( Assume File f)

|  |
| --- |
| boolean b = f.exists(); 🡪true if file exists.  boolean b = f.isFile(); 🡪true if this is a normal file.  boolean b = f.isDirectory();🡪true if f is a directory.  String s = f.getName(); 🡪name of file or directory.  boolean b = f.canRead(); 🡪true if can read file.  boolean b = f.canWrite(); 🡪true if can write file.  boolean b = f.isHidden(); 🡪true if file is hidden.  long l = f.lastModified(); 🡪Time of last modification.  long l = f.length();🡪Number of bytes in the file. |

* *Setting Attributes*

|  |
| --- |
| *f*.setLastModified(*t*); 🡪Sets last modified time to long value *t*.  boolean *b* = *f*.setReadOnly();🡪Make file read only. Returns true if successful. |

* Paths

|  |
| --- |
| String s = f.getPath();🡪path name.  String s = f.getAbsolutePath(); 🡪path name (how is it different from above?).  String s = f.getCanonicalPath(); 🡪path name. May throw IOException.  String s = f.toURL();& String s = f.toURI(); ; 🡪path with "file:" prefix and /'s. Directory paths end with /. |

* Creating and deleting files and directories

|  |
| --- |
| Boolean b = f.delete(); 🡪Deletes the file.  boolean b = f.createNewFile(); 🡪Create file, may throw IOException. true if OK; false if already exists.  boolean b = f.renameTo(f2);🡪Renames f to File f2. Returns true if successful.  boolean b = f.mkdir();🡪Creates a directory. Returns true if successful.  boolean b = f.mkdirs(); 🡪Creates directory and all dirs in path. Returns true if successful. |

|  |
| --- |
| Example: Working with File class  Import java.io.File; import java.io.IOException; public class FileTest{  public static void main(String[] args){ File f =new File("D:/Documents and Settings/esubalew/Desktop/JavaTests/Buffered.java");  System.out.println(f.exists());  **Output:**  true  true  true  Buffered.java  D:\Documents and Settings\esubalew\Desktop\JavaTests  false  true  739  System.out.println(f.canRead());  System.out.println(f.canWrite());  System.out.println(f.getName());  System.out.println(f.getParent());  System.out.println(f.isDirectory());  System.out.println(f.isFile());  System.out.println(f.length());  }  } |

## **Predefined Streams**

“java.lang” package defines a class called System, which encapsulates several aspects of the run time environment: System.out, System.in, System.err. System class contains three predefined streams, in, out and err. They are public and static fields defined inside the static final class System. System.out refers to the standard output stream. By default, this is the console. System.in refers to the standard input stream, which is the keyboard by default. System.err refers to the standard error stream, which is also the console by default**. System.in is an object of InputStream class, and System.out and System.err are objects of PrintStream class.**

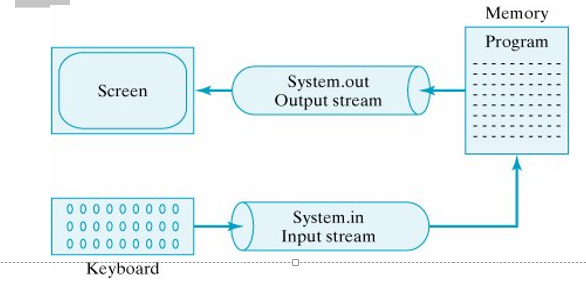


Figure 32 Predefined Input/output Streams

In java, console input is accomplished by reading from System.in. To read an input we link System.in to a InputStreamReader class as follows:

|  |
| --- |
| InputStreamReader r = new InputStreamReader (System.in); |

As InputStreamReader read single byte at a time, this affects system performance. For this purpose we use BufferReader class to read number of bytes from input buffer. To read input from buffer we wrap InputStreamReader into BufferedReader as follows:

|  |
| --- |
| BufferedReader r = new BufferedReader(new InputStreamReader ( System.in)); |

BufferedReader has a number of methods. By using this methods we can read an input from the keyboard.Some of these methods are:

* + close () 🡪Closes the stream and releases any system resources associated with it.
  + read() 🡪Reads a single character
  + readLine () 🡪 Reads a line of text.
  + ready() 🡪 Tells whether this stream is ready to be read.

# **Reference**

1. Herbert Schildt, “Java™ 2:The Complete Reference”,Fifth Edition,
2. Laura L. and Charles L. Perkins , “Teach Yourself Java in 21 days”, Copyright ©1996 by Sams.net, Publishing and its licensors, First Edition
3. Robert L.,”Object Oriented programming in C++”,
4. Rajkumar B. al., “Object Oriented Programming with Java: Essentials and Applications“, McGraw-Hill Education (India) Pvt Ltd, New Delhi, India
5. E. Balagurusamy, “Programming with Java: Premium ”, Tata McGraw-Hill Publishing Company,third Edition, New Delhi
6. ,“Introduction to Java”,
7. <https://www.w3schools.com/java/>
8. <https://www.javatpoint.com/java-oops-concepts>
9. <https://www.c-sharpcorner.com/UploadFile/3614a6/accessors-and-mutators-in-java/>