Core Java

Object Oriented Programming in Java

Lesson Objective

- Introduction to Object Orientation
- Objects and Classes
- Object Oriented Principles
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
  - Polymorphism
- Implementation of Object Oriented Principles
- Abstract Classes and Interfaces
- Inner Classes

Introduction to Object Orientation

- OOP is a paradigm of application development where programs are built around objects and their interactions with each other.
  - An Object Oriented program can be viewed as a collection of co-operating objects
- An OO program is made up of several objects that interact with each other to make up the application.
- **For example:** In a Banking System, there would be Customer objects pertaining to each customer. Each customer object would own its set of Account Objects, pertaining to the set of Savings and Current Accounts that the customer holds in the bank.
- Today, most programming languages are object oriented.
  For example: Java, C++, C#

Object Oriented Approach

- OO Approach is based on the concept of building applications and programs from a collection of “reusable entities” called “objects”.
  - Each object is capable of receiving and processing data, and further sending it to other objects.
  - Objects represent real-world business entities, either physical, conceptual, or software.
- **For example:** a person, place, thing, event, concept, screen, or report

Why Object Oriented Programming?

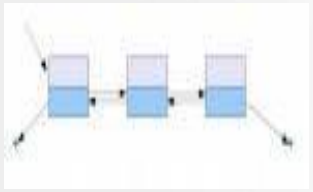
- There are problems associated with structured language, namely:
  - Emphasis is on doing things rather than on data
  - Most of the functions share global data which lead to their unauthorized access
  - More development time is required
  - Less reusability
  - Repetitive coding and debugging
  - Does not model real world well

Advantages with OOPS

- **Simplicity:** Software objects model the real world objects. Hence the complexity is reduced and the program structure is very clear.
- **Modularity:** Each object forms a “separate entity” whose internal workings are decoupled from other parts of the system.
- **Modifiability:** Changes inside a class do not affect any other part of a program, since the only “public interface” that the external world has to a class is through the use of “methods”.
- **Extensibility:** Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
- **Maintainability:** Objects can be separately maintained, thus making locating and fixing problems easier.
- **Re-usability:** Objects can be reused in different programs.

Objects

- An object is a tangible, intangible, or software entity.
- It is characterized by Identity, State, and Behavior.
    - **Identity:** It distinguishes one object from another.
    - **State:** It comprises set of properties of an object, along with its values.
    - **Behavior:** It is the manner in which an object acts and reacts to requests received from other objects



Object State

- **State** of an object is one of the possible conditions in which the object may exist.

Bank Account Modeled as a Software Object

Attributes of the object:

Account Number: A10056

Type: Savings

Balance: 40000

Customer Name: Sarita Kale

The state of an object is not defined by a “state” attribute or a set of attributes. Instead the “state” of an object gets defined as a total of all the attributes and links of that object.

Object Behavior

- **Behavior** of an object determines how an object reacts to other objects.

Behaviors of the object

- Withdraw
- Deposit
- Check Balance
- Get Monthly statement

These are the operations that the object can perform, and represents its behavior.

Through these operations or methods, an object controls its state.

Object Identity

- Two objects can posses identical attributes (state) and yet have distinct identities.

Class

- A Class characterizes common structure and behavior of a set of objects.
- It constitutes of Attributes and Operations.
- It serves as a template from which objects are created in an application.

Class name	Customer
Class attributes	Name, Address, Email-ID, TelNumber

Class operations	displayCustomerDetails()
	changeContactDetails()

Class Attributes and Operations

- Each class has a name, attributes, and operations.
  - **Attribute** is a named property of a class that describes a range of values.
  - **Operation** is the implementation of a service that can be requested from any object of the class to affect behavior.

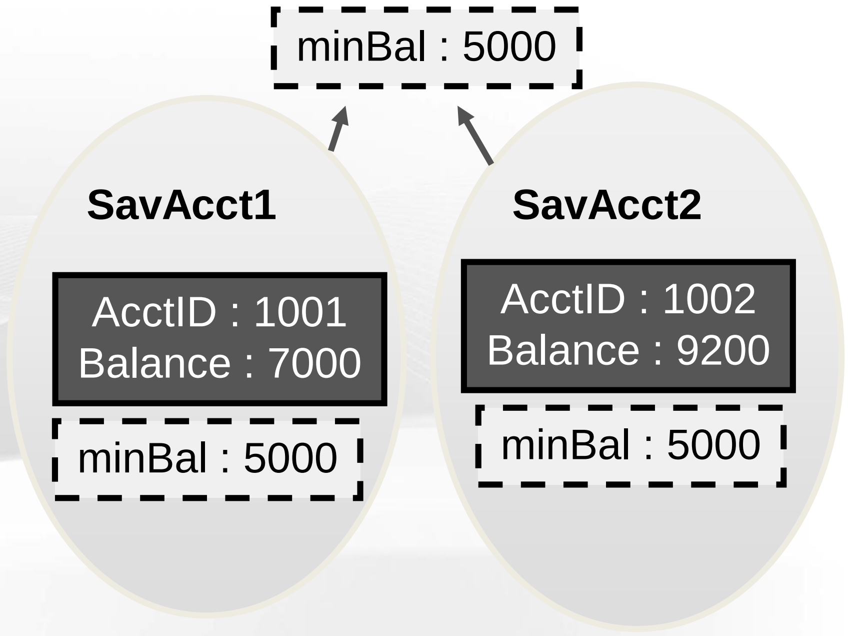
	Class Name	Account
Attributes	Class attributes	AccountNumber, Type, Balance
Operations	Class operations	withdraw(), checkBalance(), displayAccountDetails, deposit()

Constructors and Destructors

- Constructors:
  - They enable instantiation of objects against defined classes.
  - Memory is set aside for the object. Attribute values can be initialized along with object creation (if needed).
- Destructors:
  - They come into play at end of object lifetime. They release memory and other system locks.

Static Members

- Static data members are those that are shared amongst all object instances of the given type.
  - **Example:** Rate of Interest for Savings Account will be the same in the bank, so common copy is sufficient. Not so for Account Balances!

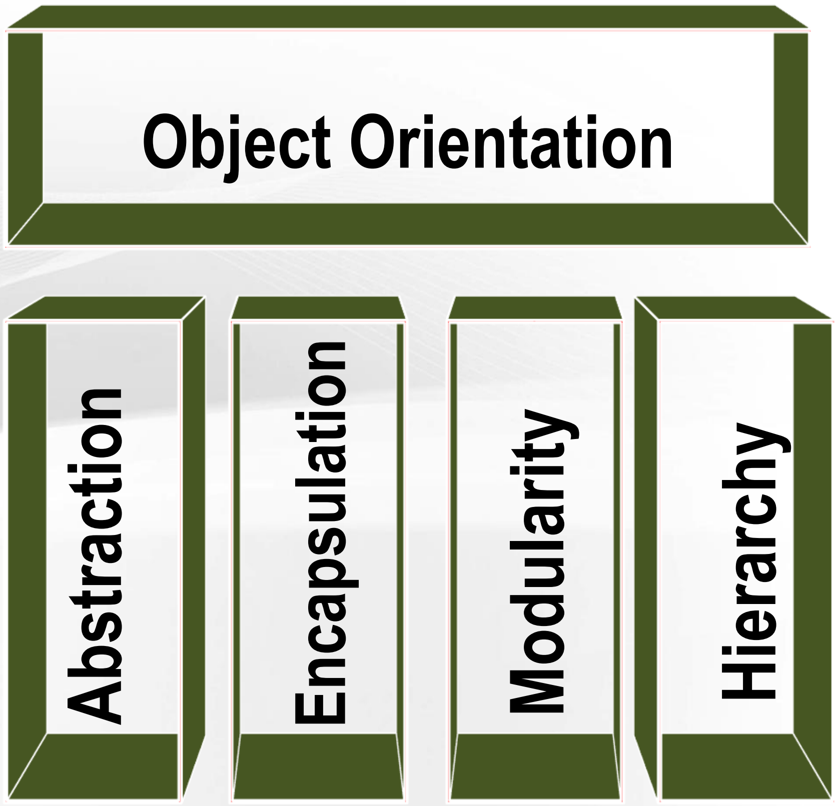


Static Members

- Static Member Functions can be invoked without an object instance.
    - **For example:** Counting the number of Customer Objects created in the Banking System – this is not specific to one object!

Object Oriented Principles

- OO is based on four basic principles, namely:
  - **Principle 1:** Abstraction
  - **Principle 2:** Encapsulation
  - **Principle 3:** Modularity
  - **Principle 4:** Hierarchy

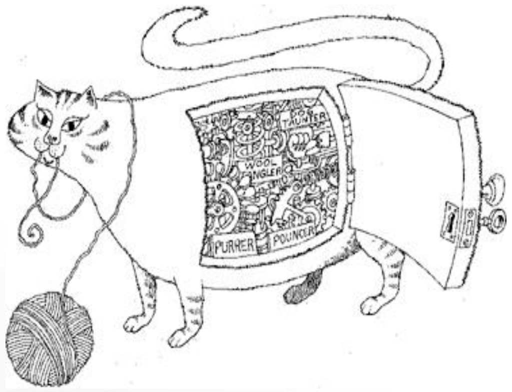


Concept of Abstraction

- Focus only on the essentials, and only on those aspects needed in the given context.
    - **For example:** Customer Height / Weight not needed for Banking System!
- Abstraction is determining the essential qualities. By emphasizing on the important characteristics and ignoring the non-important ones, one can reduce and factor out those details that are not essential, resulting in less complex view of the system.
- Abstraction means that we look at the external behavior without bothering about internal details. We do not need to become car mechanics to drive a car!

Concept of Encapsulation

- “To Hide” details of structure and implementation
  - **For example:** It does not matter what algorithm is implemented internally so that the customer gets to view Account status in Sorted Order of Account Number.
- Every object is encapsulated in such a way, that its data and implementations of behaviors are not visible to another object.
- Encapsulation allows restriction of access of internal data.



Encapsulation versus Abstraction

- Abstraction and Encapsulation are closely related.
  - Abstraction can be considered as User’s perspective.
  - Encapsulation can be considered as Implementer’s perspective.

- Why Abstraction and Encapsulation?
  - They result in “Less Complex” views of the System.
  - Effective separation of inside and outside views leads to more flexible and maintainable systems.

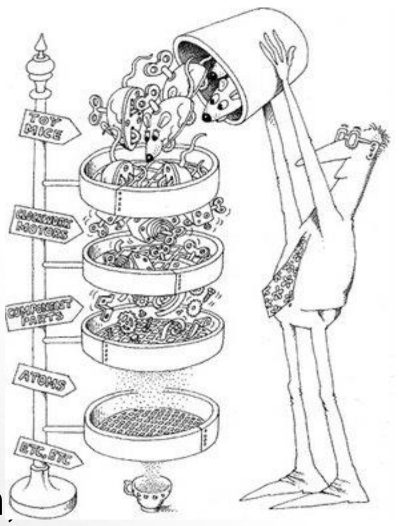
Concept of Modularity

- Decomposing a system into smaller, more manageable parts
  - **Example:** Banking System can have different modules to take care of Customer Management, Account Transactions, and so on.
- Why Modularity?
  - Divide and Rule! Easier to understand and manage complex systems.
  - Allows independent design and development of modules.

- As modules are groups of related classes, it is possible to have parallel developments of modules. Changes in one may not affect the other modules. Modularity is an essential characteristic of all complex systems. Well designed modules can be reused in similar situations in other designs.

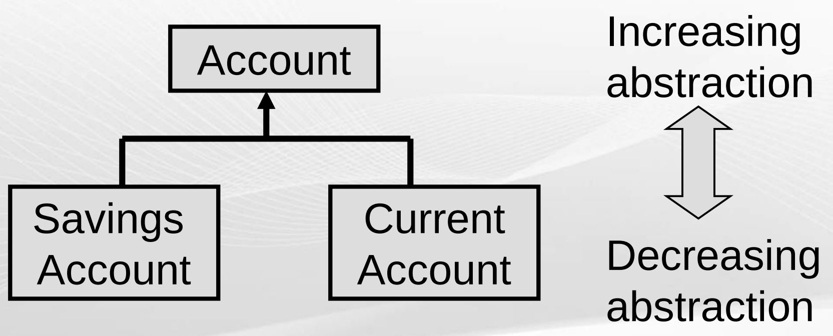
Concept of Hierarchy

- A ranking or ordering of abstractions on the basis of their complexity and responsibility
- It is of two types:
  - **Class Hierarchy:** Hierarchy of classes, **Is A** Relationship.
    - Example: Accounts Hierarchy
  - **Object Hierarchy:** Containment amongst Objects, **Has A** Relationship.
    - Example: Window has a Form seeking customer information, which has text boxes and various buttons.



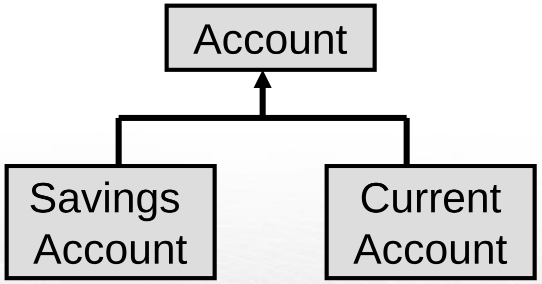
Inheritance

- Inheritance
  - It is a powerful technique that enables code reuse resulting in increased productivity, and reduced development time.
  - It allows for designing extensible software.

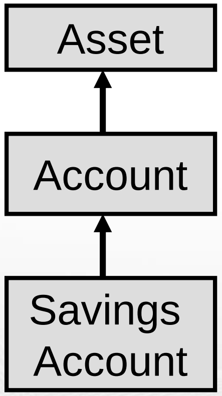


- Inheritance is the process of creating new classes, called *derived classes*, from existing or base classes. The derived class inherits all the capabilities of the base class, but can add some specificity of its own. The base class is unchanged by this process.

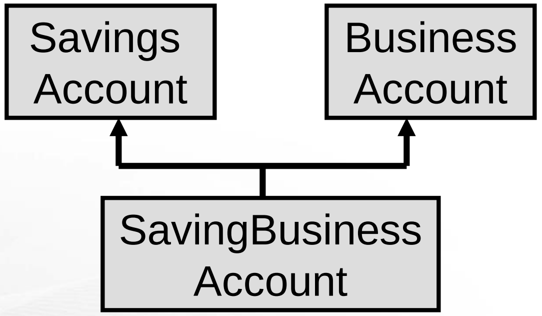
Types of Inheritance



Single-level inheritance

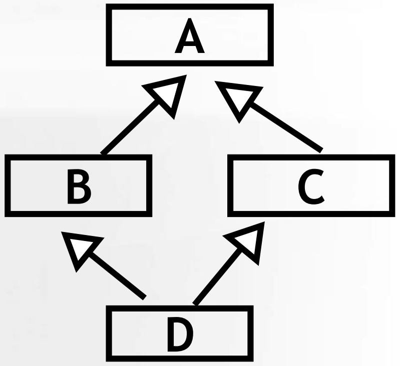


Multilevel inheritance



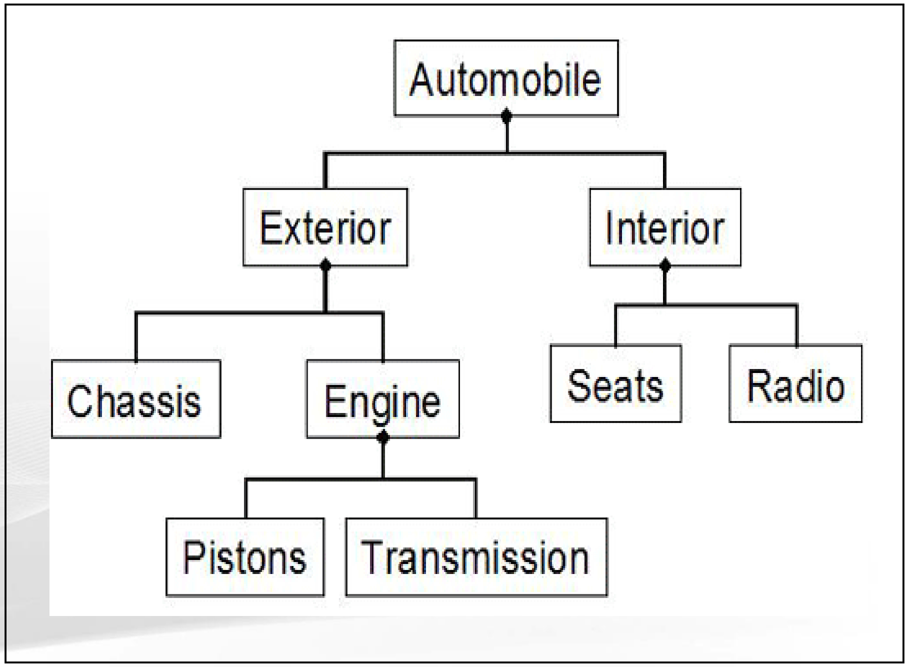
Multiple inheritance

Multiple inheritance challenges: A name conflict introduced by a shared super-class (A) of super-classes (B and C) used with "multiple inheritance".



Object Hierarchy

- Composition
- "Has-a" hierarchy is a relationship where one object "belongs" to (is a part or member of) another object, and behaves according to the rules of ownership.



Polymorphism

- It implies One Name, Many Forms.
- It is the ability to hide multiple implementations behind a single interface.
- Polymorphism allows different objects to respond to the same message in different ways!
- There are two types of Polymorphism, namely:
  - Static Polymorphism
  - Dynamic Polymorphism

Polymorphism

- Static Polymorphism:
  - The “Form” can be resolved at compile time, achieved through **overloading**.
    - For example: An operation “Sort” can be used for sorting integers, floats, doubles, or strings.
- Dynamic Polymorphism:
  - The “Form” can be resolved at run time, achieved through **overriding**.
    - For example: An operation “calculateInterest” for Accounts, where an Account can be of different types like Current or Savings.

Class and Object Implementation

```
public class Employee {
        private String name; // instance variables
        private String tel;

        public Employee(String n, String t){  //constructor
                name = n;
                tel  = t;
        }
        public void printinfo() {
          System.out.println(name+"'s number is "+tel);

        }

      public static void main(String[] args) // Entry Point
        {
               //Object Creation
               Employee gao=new Employee("Gao", "4548");
                 gao.printinfo();
        }

    }
```

Inheritance Implementation

```
public class Faculty extends Employee {
        private String secName;
        public Faculty(String n, String t, String s) {
                super(n, t);                              //have to be the first line  constructor
                secName = s;
        }
        public void changeSec (String s) {
                secName = s;
        }
        public void printinfo() {
                super.printinfo();
                System.out.println("Secretary's name is " + secName);
        }
        public static void main(String[] args) {
                Faculty lixin=new Faculty("Lixin", "4548", "June");
                lixin.printinfo();
        }
}
```

Function Overloading

- Two or more methods within the same class share the *same* name.
- They should differ either in number of arguments or types of arguments.
- Java implements compile-time polymorphism by:
  - Overloading Constructors
  - Overloading Normal Methods

Function Overloading

```
public class Calculator {

       public int add(int x, int y){
              return(x+y);
       }
       public int add(int x,int y, int z) {
               return(x+y+z);
       }

      public static void main(String[] args)
       {
              Calculator c=new Calculator();
              int d=c.add(10,20);
              int e=c.add(10,20,30);
       }
   }
```

Function Overriding

- In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method.
- Overridden methods allow Java to support run-time polymorphism.
- Base class and derived class method has same signature.
- Final methods can’t be overriden

Function Overriding

```
public class Calculator_New extends Calculator {

        public int add(int x, int y){
                x++;
                y++;
                return(x+y);
        }
    public static void main(String[] args)
        {
                Calculator c;
                c=new Calculator_New();
                int d=c.add(10,20);            //32
                 c=new Calculator();
                int e=c.add(10,20);           //30
        }
    }
```

Abstract Classes

- Abstract Method
  - Methods do not have implementation.
  - Methods declared in interfaces are always abstract.
    - Example:

```
abstract int  add(int a, int b);
```

- Abstract class
  - Provides common behavior across a set of subclasses.
  - Not designed to have instances that work.
  - One or more methods are declared but may not be defined
  - Advantages:
    - Code reusability.
    - Help at places where implementation is not available.

Abstract Classes

- Declare any class with even one method as abstract as *abstract*.
- Cannot be instantiated.
- Use *Abstract* modifier for:
  - Constructors
  - Static methods
- Abstract class' subclasses should implement all methods or declare themselves as *abstract*.
- Can have concrete methods also.

Abstract classes

```
abstract class AccountAbstract {
       public int balance;
        abstract public void deposit(int amt);              //abstract method
        public void display()
       {    System.out.println("Balance = "+balance);
        }
  }
  public class Saving extends AccoutAbstract {
     public void deposit(int amt){
              balance=balance +amt;
       }
     public static void main(String[] args)
        {
               Saving s=new Saving();
              s.deposit(500);
       }
    }
```

Interface

- Interface defines a data-type without implementation.
- The interface approach is sometimes known as programming by contract.
- It’s essentially a collection of constants and abstract methods.
- An interface is used via the keyword “implements”. Thus, a class can be declared as follows:

```
class MyClass implements MyInterface{
  …   }
```

Interfaces

- A Java interface definition looks like a class definition that has only abstract methods, although the abstract keyword need not appear in the definition

```
public interface Testable {
void method1();
void method2(int i, String s);
int x=10;
}
```

note no bodies for the methods, public by default

Static final variable

Declaring and Using Interfaces

```
    public interface simple_cal {
      int add(int a, int b);
      int i=10;
 }
//Interfaces are to be implemented.
class calci implements simple_cal {
      int add(int a, int b){
                  return a+b;
      }
}
```

Interface - Rules

- Methods in an interface are always public and abstract.
- Data members in a interface are always public, static and final.
- Interface is a purely abstract class; has only signatures and no implementation.
- Interface members are implicitly public abstract.
- Interface members must not be static.
- Interfaces can extend other interfaces.
- A class can inherit from a single base class, but can implement multiple interfaces.

Interfaces and Abstract Classes

Abstract classes	Interfaces
Abstract classes are used only when there is a "is-a" type of relationship between the classes.	Interfaces can be implemented by classes that are not related to one another.
You cannot extend more than one abstract class.	You can extend more than one interface.
Abstract class can contain abstract as well as implemented methods.	Interfaces contain only abstract methods.
With abstract classes, you grab away each class's individuality.	With Interfaces, you merely extend each class's functionality.

Nested Classes

- Class within another class.
- Scope is bounded by the scope of the enclosing class.
- Use to reflect and enforce the relationship between two classes.

```
    class EnclosingClass{
    . . .
    class ANestedClass {              //known only within Enclosing class
        . . .
    }
}
```

Types of Nested Classes

- Nested classes are of two types:
  - Static:
    - Access members of the enclosing class through an object.
    - Cannot refer to members of the enclosing class directly.
  - Non-static:
    - Also called as *Inner* classes.
    - Access to all members, including private members, of the class in which they are nested.
    - Can refer to the members of enclosing class in the same way as non-static members.

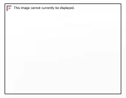
Inner class Example

```
class Outer{
int intOuter = 100;
void Test(){
    Inner inObj = new Inner();
    inObj.display();
}
class Inner{
  void Display(){
  System.out.println(“Value of outer variable = “ + intOuter);
}   } // end of class Inner
}// end of class Outer
class Demo{
public static void main(String args[]){
Outer outObj = new Outer();
outObj.test();
}
 }
```

Output: Value of outer variable =100

Anonymous Inner Class

- Do not have a name.
- Defined at the location they are instantiated using additional syntax with the *new* operator.
- Used to create objects "on the fly" in contexts such as:
    - Method return value.
    - Argument in a method call.
    - Variable initialization.



Final Variable

- Variable declared as *final* acts as a constant.
- Once initialized, it’s value cannot be changed.
- Example:
  - final int i = 10;

Final Method

- Method declared as final cannot be overridden in subclasses.

- Their values cannot change their value once initialized.
  - Example:

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

Final Class

- Cannot be sub-classed at all.
- Examples: *String* and *StringBuffer* classes in Java.
- A class or method cannot be abstract and final at the same time.

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

Packages

- Packages are used to group related classes and interfaces.
- Packages are useful method of grouping classes to avoid name clashes.
- Classes with same name can be put into different packages.
- A *package* is a collection of related types providing access protection.

Packages

- You should bundle these classes and the interface in a package for several reasons:

- You and other programmers can easily determine that these types are related.
- You and other programmers know where to find types that provide graphics-related functions.
- The names of your types won't conflict with types names in other packages, because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Creating package

- Create the class or interface.
- Put the first statement of that class or inteface as below.

  package <package_name>;

  Example : package pack1;
- compile the above class or interface which will generates .class file.
- Create folder with the package name and put the class file in the folder.
- If you are having sub packages the corresponding sub folders needs to be created

Create a package

```
package pkg1;
public class Student
{
    public int sno;
    public void getSno()
    {
    }
}
```

Using Packages

- Write the import statement in the file where you want use the package classes or interfaces.

  Example: import pkg1.Student;

- Set the classpath from command prompt to get package available to you.

  Example :

  Set classpath=.;d:\assignments;

  Assuming that pkg1 folder is in side the “d:\assignment” folder

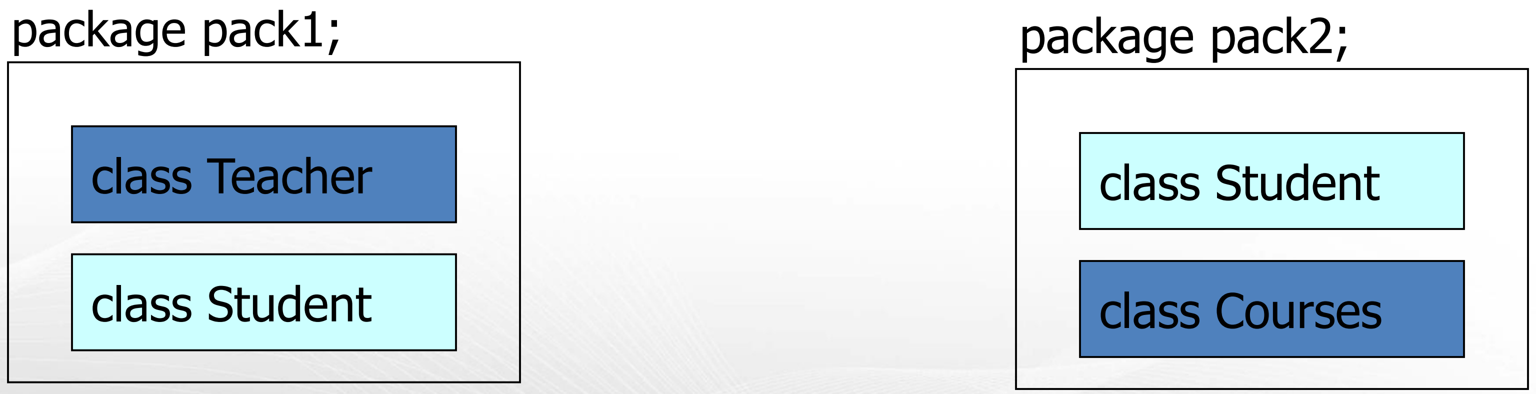
- Running application whose main class is in package

  javac pkg1.Student;

Using package

```
 import pkg1.*;

class MainClass
{
    public static void main(String a[])
{
   Student s1=new Student();
    s1.getStudent();
}
}
```

Packages and Name Space Collision



- Namespace collision can be avoided by accessing classes with the same name in multiple packages by their fully qualified name.

Commonly used Java Packages

Package Name	Description
java.lang	Classes that apply to the language itself, which includes the Object class, the String class, and the System class. It also contains the special classes for the primitive types (Integer, Character, Float, and so on). Classes belonging to java.lang package need not be explicitly imported.
java.util	Utility classes, such as Date, as well as collection classes, such as Vector and Hashtable
java.io	Input and output classes for writing to and reading from streams (such as standard input and output) and for handling files
java.net	Classes for networking support, including Socket and URL (a class to represent references to documents on the World Wide Web)
java.applet	Classes to implement Java applets, including the Applet class itself, as well as the AudioClip interface

Static Import

- Static import enables programmers to import static members.
- Class name and a dot (.) are not required to use an imported static member.

```
import static java.lang.Math.*;
public class StaticImportTest
{
 public static void main( String args[] )
 {
 System.out.printf( "sqrt( 900.0 ) = %.1f\n", sqrt( 900.0 ) );
 } // end main
}
```

Note: It’s not Math.sqrt

Core Java

Exception Handling in Java

Lesson Objective

- What is Exception?
- Why Exception Handling?
- Java Exception Class Hierarchy
- Handle Exception in Java
    - Using try and catch
    - Multiple catch
    - Finally Clause
    - Throwing an Exception
    - Throws Clause
- Create your own Exceptions

What is Exception?

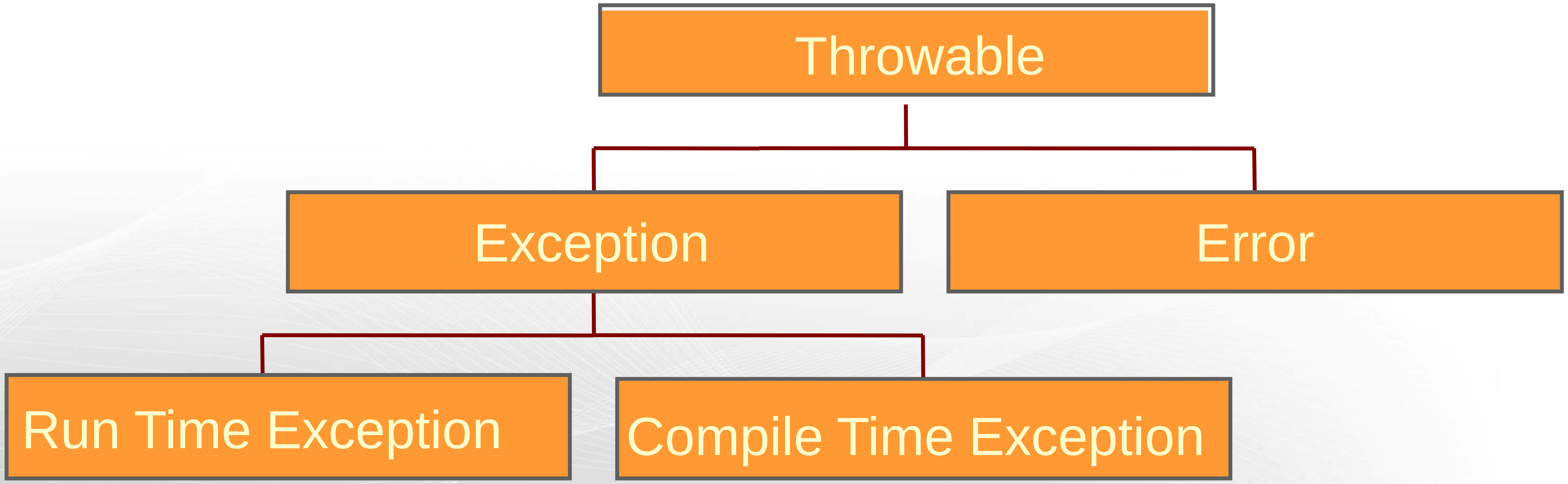
- Are abnormal events that might occur during program execution
- They terminate program execution abruptly
- Such abnormal events have to be handled to prevent the execution of the program from being terminated abruptly
- Examples:
  - Hard disk crash;
  - Out of bounds array access;
  - Divide by zero, and so on

Why Exception Handling?

- No matter how well-designed a program is, there is always a chance that some kind of error will arise during its execution, for example:
  - Attempting to divide by 0
  - Attempting to read from a file which does not exist
  - Referring to non-existing item in array
- Programmer should always be prepared for the worst.
- The preferred way of handling such conditions is to use exception handling, an approach that separates a program's normal code from its error-handling code.

DefaultDemo.java

Java Exception Class Hierarchy



Unchecked Exception Checked Exception

Error

- An **Error** is a subclass of **Throwable** that indicates serious problems that a reasonable application should not try to catch
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
- These are rare and usually fatal and therefore not supposed to be handled by the program.
- Instances of error are thrown, when the Java Virtual Machine faces some memory leakage problem, insufficient memory problem, dynamic linking failure or when some other "hard" failure in the virtual machine occurs.
- Obviously, in case of an error, the program stops executing.

Exception

- The **Exception** class and its subclasses are a form of **Throwables.** They indicate conditions, which a reasonable application may want to catch.

Exception Types

- **Checked Exception**
  - They are checked by the compiler at the time of compilation.
  - They must be handled in your code, or passed to parent classes for handling.
  - Some examples of **Checked exceptions** include:
    - IOException, SQLException, ClassNotFoundException

- **UnChecked Exception**
  - It is called **unchecked exception** because the compiler does not check to see if a method handles or throws these exceptions.
  - Example : ArithmeticException, ArrayIndexOutOfBoundsException

Handling Exception

- Using try and catch
- Multiple catch
- Finally Clause
- Throwing an Exception
- Throws Clause

Keywords for Exception Handling

- try
  - **This marks the start of a block associated with a set of exception handlers.**
- catch
  - **The control moves here if an exceptions is generated.**
- finally
  - **This is called irrespective of whether an exception has occurred or not.**
- throws
  - **This describes the exceptions which can be raised by a method.**
- throw
  - **This raises an exception to the first available handler in the call stack, unwinding the stack along the wayBehavior** of an object determines how an object reacts to other objects

Try and Catch

- The **try** structure has three parts:
  - The **try** block
    - Code in which exceptions are thrown
  - One or more **catch** blocks
    - To respond to various types of Exceptions
  - An optional **finally** block
    - Code to be executed last under any circumstances
- The **catch** Block:
  - If a line in the **try** block causes an exception, program flow jumps to the **catch** blocks.
  - If any **catch** block matches the exception that occurred that block is executed.

Using Try and Catch

```
try {
            // code in which exceptions may be thrown
   } catch (ExceptionType1  identifier) {
            // code executes if an ExceptionType1 occurs
   } catch (ExceptionType2 identifier) {
            // code executes if an ExceptionType2 occurs
   } finally  {
            // code executed last in any case
   }
```

Using Try and Catch

```
class DefaultDemo {
    public static void main(String a[]) {
      String str = null;
      try {
              str.equals(“Hello”);
         } catch(NullPointerException ne) {
                     str = new String(“Hello”);
                     System.out.println(str.equals(“Hello”));
                 }
        System.out.println(“Continuing in the program…..”); }
 }
```

TryCatchDemo.java

Multiple Catch Blocks

- If you include multiple **catch** blocks, the order is important.
- You must catch subclasses before their ancestors.

```
public void divide(int x,int y)
{
    int ans=0;
    try{
            ans=x/y;
    }catch(Exception e) { //handle }
    catch(ArithmeticException f) {//handle}  //error
```

MultiCatch.java

Nested Try Catch Block

```
try {
      int a = arg.length;
      int b = 10 / a;
      System.out.println("a = " + a);
    try {
      if(a==1)
      a = a/(a-a);
      if(a==2) {
         int c[] = { 1 };
         c[42] = 99;
                  }
         } catch(ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out-of-bounds: " + e); }
  } catch(ArithmeticException e) {
          System.out.println("Divide by 0: " + e); }
```

Finally Clause

- The **finally** block is optional.
- It is executed whether or not exception occurs.

FinallyDemo.java

```
public void divide(int x,int y)
{
    int ans;
    try{
    ans=x/y;
    }catch(Exception e) {  ans=0;  }
    finally{
System.out.println(“Task Completed”);  //  always executed
}
}
```

Throwing an Exception

- You can throw your own runtime errors:
  - To enforce restrictions on use of a method
  - To "disable" an inherited method
  - To indicate a specific runtime problem
- To throw an error, use the **throw** Statement
  - throw ThrowableInstance
- **ThrowableInstance** is any **Throwable** Object

Throwing an Exception

```
class ThrowDemo {
void proc() {
try {
    throw new ArithmeticException(“From Exception”);
} catch(ArithmeticException e) {
    System.out.println("Caught inside demoproc.");
    throw e; // rethrow the exception
} }
public static void main(String args[]) {
ThrowDemo t=new ThrowDemo();
try {
    t.proc();
} catch(ArithmeticException e) {
System.out.println("Recaught: " + e); } } }
```

Using The Throws Clause

- If a method might throw an exception, you may declare the method as “throws” that exception and avoid handling the exception yourself.

```
class ThrowsDemo {
public static void main(String args[]) {
try {
doWork();
} catch (ArithmeticException e) {
System.out.println("Exception: " + e.getMessage());
} }
static void doWork() throws ArithmeticException {
int array[] = new int[100];
array[100] = 100;
} }
```

User Defined Exceptions

- Write a class that extends(indirectly) Throwable.
- What Superclass to extend?
  - For unchecked exceptions: RuntimeException
  - For checked exceptions:
    - Any other Exception subclass or the Exception itself

```
class AgeException extends Exception {
private int age;
AgeException(int a) {
age = a;
}
public String toString() {
return age+" is an invalid age"; } }
```

UserException.java

ExceptionDemo.java

Core Java

Collections

Lesson Objective

- The Collection API
- Collection Interface
- Generics
- Enhanced For loop
- Auto-boxing with Collections
- Implementing classes
- The Legacy Classes and Interfaces
- Arrays and Collections

Collections Framework

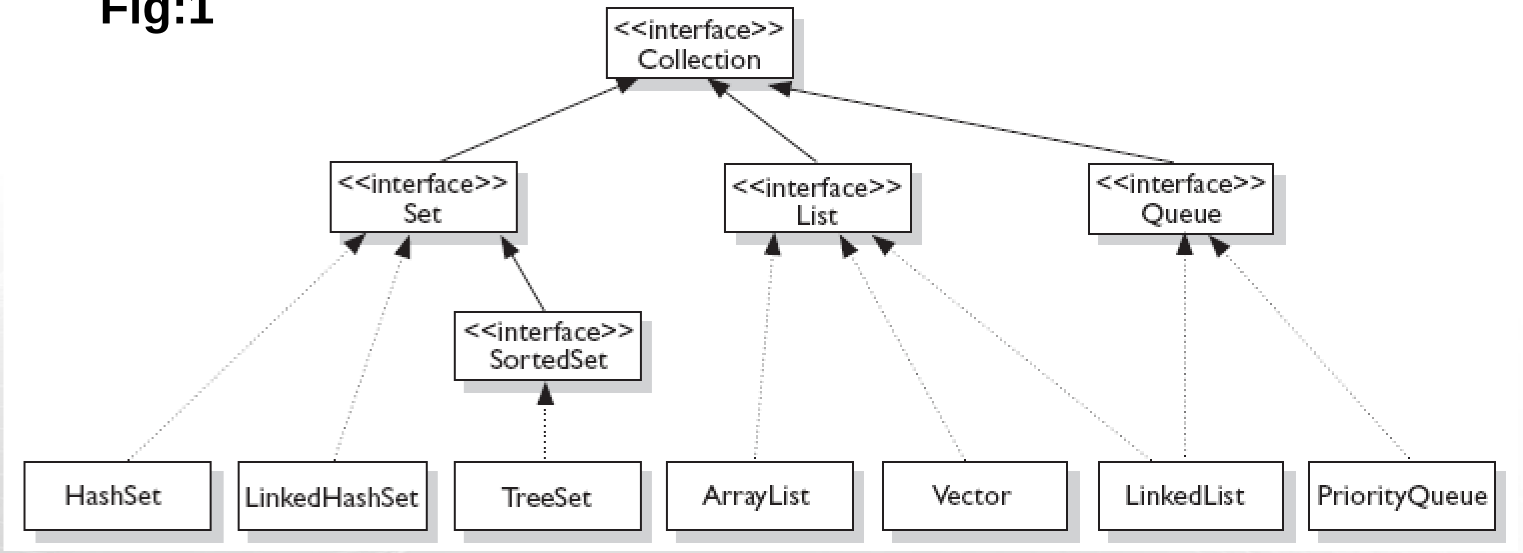
- A Collection is a group of objects.
- Collections framework provides a a set of standard utility classes to manage collections.
- Collection is used to store, retrieve objects, and to transmit them from one method to another.
- Collections Framework consists of three parts:
  - Core Interfaces
  - Concrete Implementation
  - Algorithms such as searching and sorting

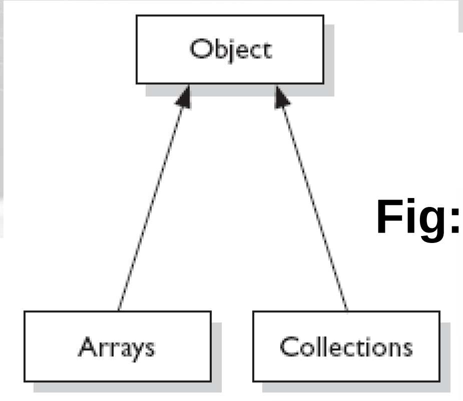
Advantages of Collections

- **Reduces programming effort** by providing useful data structures and algorithms so you do not have to write them yourself.
- **Increases performance** by providing high-performance implementations of useful data structures and algorithms.
- **Provides interoperability between unrelated APIs** by establishing a common language to pass collections back and forth.
- **Reduces the effort required to learn APIs** by eliminating the need to learn multiple ad hoc collection APIs.
- **Reduces the effort required to design and implement APIs** by eliminating the need to produce ad hoc collections APIs.
- **Fosters software reuse** by providing a standard interface for collections and algorithms to manipulate them.

Interfaces and Implementations

Fig:1





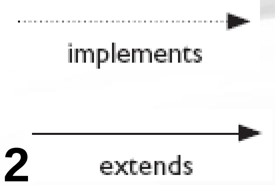
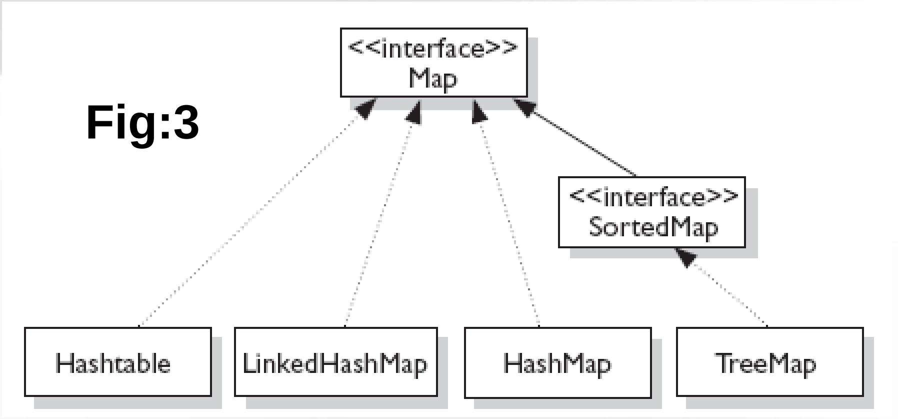


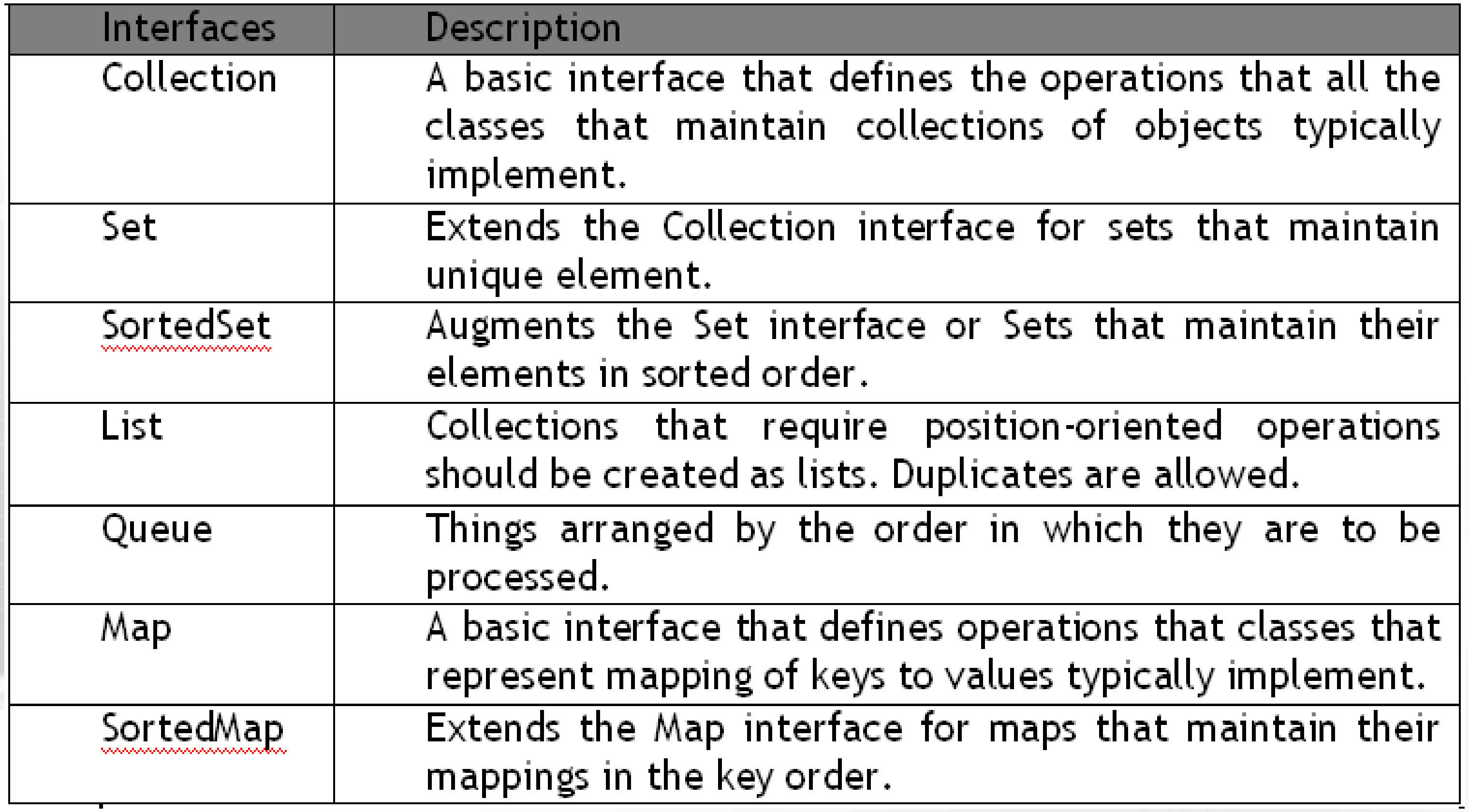
Fig:2

Fig:3



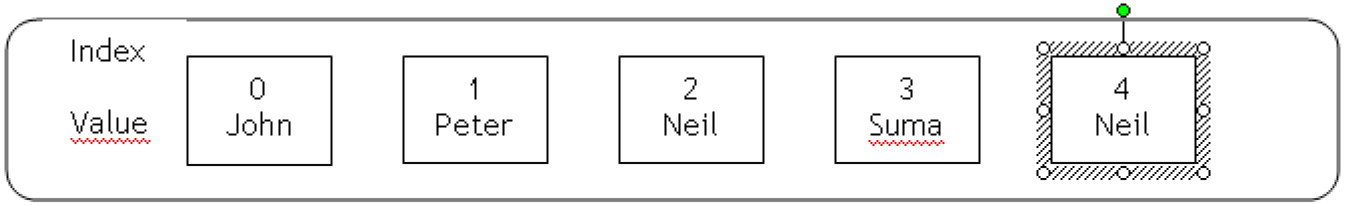
Collection Interfaces

- Let us discuss some of the collection interfaces:

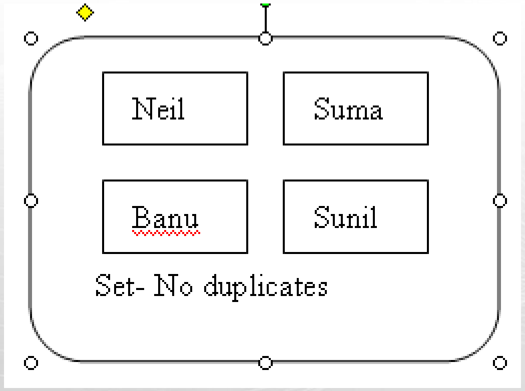


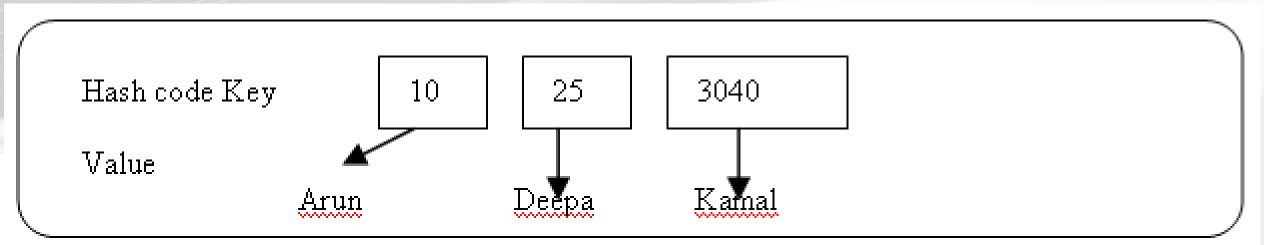
Interfaces	Description
Collection	A basic interface that defines the operations that all the classes that maintain collections of objects typically implement.
Set	Extends the Collection interface for sets that maintain unique element.
SortedSet	Augments the Set interface or Sets that maintain their elements in sorted order.
List	Collections that require position-oriented operations should be created as lists. Duplicates are allowed.
Queue	Things arranged by the order in which they are to be processed.
Map	A basic interface that defines operations that classes that represent mapping of keys to values typically implement.
SortedMap	Extends the Map interface for maps that maintain their mappings in the key order.

Illustration of List, Set, and Map



List - Duplicates allowed





Hash Map - Key generated from RollNo

Collection Implementations

		Implementations				
		Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Interfaces	Set	HashSet		TreeSet		LinkedHashSet
	List		ArrayList		LinkedList	
	Map	HashMap		TreeMap		LinkedHashMap

Collection Implementations

- **HashSet**: A HashSet is an unsorted, unordered Set.

  It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation is, the better access performance you will get.

  Use this class when you want a collection with no duplicates and you do not care about order when you iterate through it.

- **LinkedHashSet**: A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

  Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

Collection Implementations

- **TreeSet**: A TreeSet stores objects in a sorted sequence. It stores its elements in a tree and they are automatically arranged in a sorted order.

- **ArrayList**: Think of this as a **growable** array. It gives you fast iteration and fast random access. It is an ordered collection (by index).

  However, it is not sorted. ArrayList now implements the new RandomAccess interface — a marker interface (meaning it has no methods) that says, “this list supports fast (generally constant time) random access.”

  Choose this over a **LinkedList** when you need fast iteration but are not as likely to be doing a lot of insertion and deletion.

- **LinkedList**: A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.

Collection Implementations

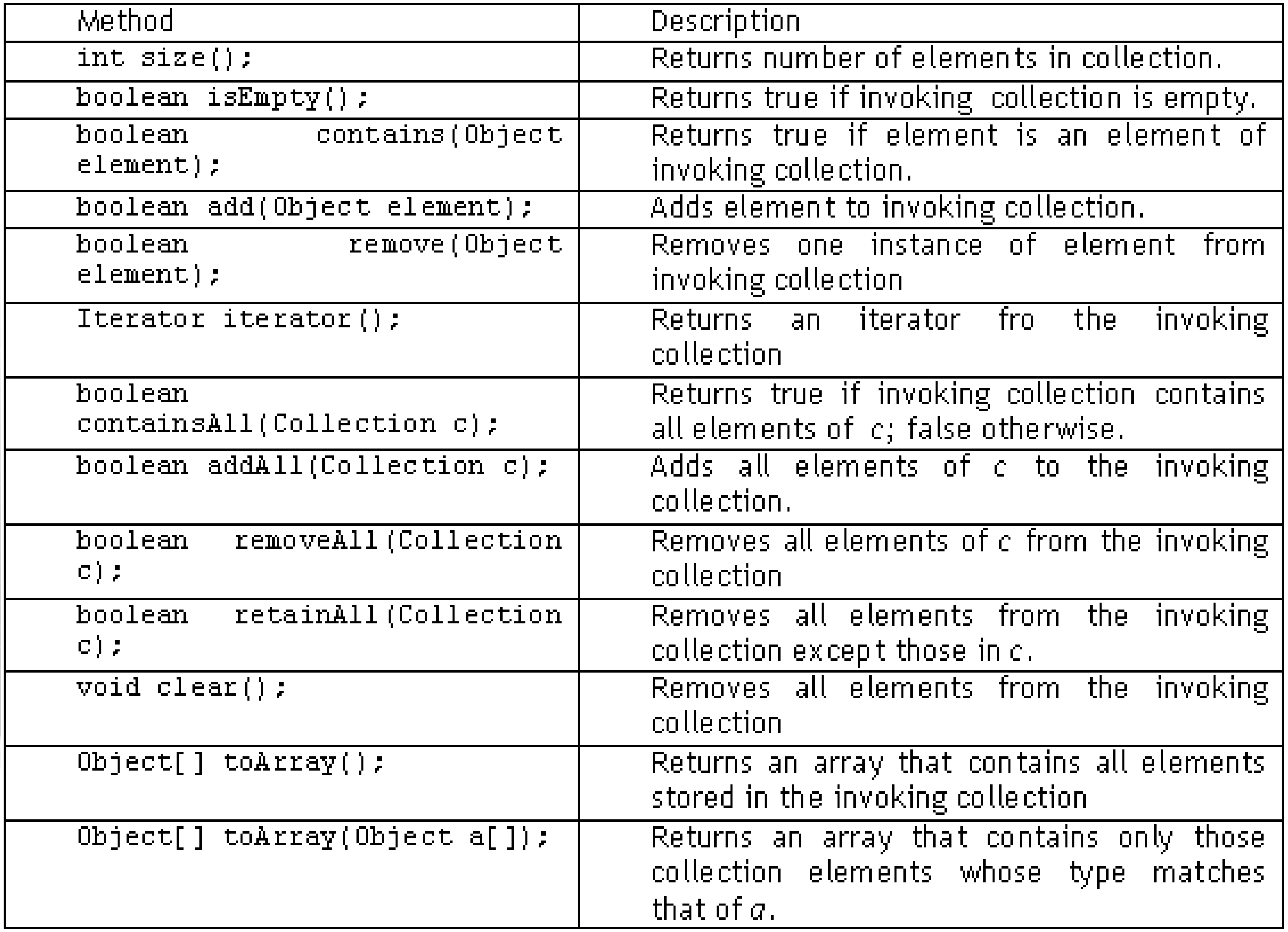
- **HashMap:** The HashMap gives you an unsorted, unordered Map. When you need a Map and you do not care about the order (when you iterate through it), then HashMap is the way to go.

  The other maps add a little more overhead. Where the keys land in the Map is based on the key’s hashcode. So, like HashSet, the more efficient your hashCode() implementation, the better access performance you will get. HashMap allows one null key and multiple null values in a collection.

- **TreeMap:** TreeMap is a sorted Map.

- **LinkedHashMap:** Like its Set counterpart, LinkedHashSet, the LinkedHash-Map collection maintains insertion order (or, optionally, access order). Although it will be somewhat slower than HashMap for adding and removing elements, you can expect faster iteration with a LinkedHashMap.

Collection Interface methods



Method	Description
`int size();`	Returns number of elements in collection.
`boolean isEmpty();`	Returns true if invoking collection is empty.
`boolean contains(Object element);`	Returns true if element is an element of invoking collection.
`boolean add(Object element);`	Adds element to invoking collection.
`boolean remove(Object element);`	Removes one instance of element from invoking collection
`Iterator iterator();`	Returns an iterator fro the invoking collection
`boolean containsAll(Collection c);`	Returns true if invoking collection contains all elements of *c*; false otherwise.
`boolean addAll(Collection c);`	Adds all elements of c to the invoking collection.
`boolean removeAll(Collection c);`	Removes all elements of c from the invoking collection
`boolean retainAll(Collection c);`	Removes all elements from the invoking collection except those in c.
`void clear();`	Removes all elements from the invoking collection
`Object[] toArray();`	Returns an array that contains all elements stored in the invoking collection
`Object[] toArray(Object a[]);`	Returns an array that contains only those collection elements whose type matches that of *a*.

Demo :ArrayListDemo

Demo:

ArrayListSemo.java

ArrayListDemo.java

Iterators

- **Iterator** is an object that enables you to traverse through a collection.
- It can be used to remove elements from the collection selectively, if desired.

```
public interface Iterator<E>
 {
 boolean hasNext();     //returns true if there are more elements
 E next();              // returns next element
void remove();           // remove element from collection
 }
```

Demo :Iterators

Demo:

ItTest.java

ItTestjava

Comparator Interface

- The **java.util.Comparator** interface can be used to sort the elements of an Array or a list in the required way.

- It gives you the capability to sort a given collection in any number of different ways.

- Methods defined in Comparator Interface are as follows:
  - int compare(Object o1, Object o2)
    - *obj1* and *obj2* are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned.
  - boolean equals(Object obj)
    - *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**.

Demo :Comparator

Demo:

ComparatorExample.java

ComparatorExample.java

Generics

- **Generics** is a mechanism by which a single piece of code can manipulate many different data types without explicitly having a separate entity for each data type.

  - Before Generics:

```
List myIntegerList = new LinkedList(); // 1
myIntegerList.add(new Integer(0)); // 2
Integer x = (Integer) myIntegerList.iterator().next(); // 3
```

    - **Note:** Line no 3 if not properly typecasted will throw runtime exception

  - After Generics:

```
List<Integer> myIntegerList = new LinkedList<Integer>(); // 1
myIntegerList.add(new Integer(0)); //2
Integer x = myIntegerList.iterator().next(); // 3
```

Why Generics?

- Problem: Collection element types:
  - Compiler is unable to verify types.
  - Assignment must have type casting.
  - ClassCastException can occur during runtime.

- Solution: Generics
  - Tell the compiler type of the collection.
  - Let the compiler fill in the cast.
    - **Example:** Compiler will check if you are adding Integer type entry to a String type collection (compile time detection of type mismatch).

Using Generics

- You can instantiate a generic class to create type specific object.
  - In J2SE 5.0, all collection classes are rewritten to be generic classes.
  - Example:

```
Vector<String> vs = new Vector<String>();
vs.add(new Integer(5)); // Compile error!
vs.add(new String(“hello”));
String s = vs.get(0); // No casting needed
```

Using Generics

- Generic class can have multiple type parameters.
- Type argument can be a custom type.
  - Example:

```
HashMap<String, Mammal> map =
   new HashMap<String, Mammal>();
   map.put(“wombat”, new Mammal("wombat"));
   Mammal w = map.get(“wombat”);
```

Enhanced for loop

- Iterating over collections looks cluttered:

```
void printAll(Collection<emp> e) {
 for (Iterator<emp> i = e.iterator(); i.hasNext(); )
 System.out.println(i.next()); } }
```

- Using enhanced **for loop**, we can do the same thing as:

```
void printAll(Collection<emp> e) {
 for (emp t : e) )
 System.out.println(t); }}
```

  - When you see the colon (:) read it as "in."
  - The loop above reads as "for each emp t in e."

- The enhanced for loop can be used for both Arrays and Collections

Implementing Classes

- ArryaList
- HashSet
- TreeSet
- HashMap

ArrayList Class

- An ArrayList Class can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.
- It gives faster Iteration and fast random access.
- It uses Ordered Collection (by index), but not Sorted.

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

Demo : ArrayList Class

Demo:

ArrayListTest.java

ArrayListDemo.java

HashSet Class

- HashSet Class does not allow duplicates.
- A HashSet is an unsorted, unordered Set.
- It can be used when you want a collection with no duplicates and you do not care about the order when you iterate through it.

Demo : HashSet Class

Demo:

SetTest.java

SetTest.java

TreeSet Class

- TreeSet does not allow duplicates.
- It iterates in sorted order.
- Sorted Collection:
  - By default elements will be in ascending order.
- Not synchronized:
  - If more than one thread wants to access it at the same time, then it must be synchronized externally.

Demo : TreeSet Class

Demo:

TreesetDemo.java

TreesetDemo.java

HashMap Class

- Map is an object that stores ***key/value*** pairs.
- Given a key, you can find its value. Keys must be unique, however values may be duplicated
- HashMap uses the hashcode value of an object to determine how the object should be stored in the collection.
- Hashcode is used again to help locate the object in the collection.
- HashMap gives you an unsorted and unordered Map.
- It allows one null key and multiple null values in a collection.

Demo : HashMap Class

Demo:

HashMapDemo.java

HashMapDemo.java

Vector Class

- The **java.util.Vector** class implements a growable array of Objects.
- It is same as ArrayList. However, Vector methods are synchronized for thread safety.
- New java.util.Vector is implemented from List Interface.
- Creation of a Vector:
  - Vector v1 = new Vector(); // allows old or new methods
  - List v2 = new Vector(); // allows only the new (List) methods.

Demo : Vector Class

Demo:

VectorDemo.java

VectorDemo.java

Hashtable Class

- It is a part of **java.util package**.
- It implements a hashtable, which maps keys to values.
    - Any non-null object can be used as a key or as a value.
    - The Objects used as keys must implement the **hashcode** and the **equals method.**
- Synchronized class
- The Hashtable class only stores objects that override the **hashCode()** and **equals()** methods that are defined by Object.

Demo : HashTable Class

Demo:

HashTableDemo.java

HashTableDemo.java

Arrays Class

- The java.util.Arrays class is basically a set of static methods that are all useful for working with arrays.
- The Arrays class contains various methods for manipulating arrays such as sorting and searching
- In addition to that, it has got many utility methods for using with arrays such as a method for viewing arrays as lists and methods for printing the contents of an array, whatever be the dimension of the array.

Demo : Arrays Class

Demo:

ArraysMethodTester.java

ArraysMethodTester.java

Collections Class

- The java.util.Collections class is basically a set of static methods which operates on Collection.
- The Collections class contains various methods for manipulating collections such as reverse, sort, max, min, binarySearch etc.

Demo : Collections Class

Demo:

CollectionsDemo.java

CollectionsDemo.java

Common Best Practices on Collections

- Let us discuss some of the best practices on Collections:
  - Use for-each liberally.
  - Presize collection objects.
  - Note that Vector and HashTable is costly.
  - Note that LinkedList is the worst performer.
  - Choose the right Collection.
  - Note that adding objects at the beginning of the collections is considerably slower than adding at the end.
  - Encapsulate collections.
  - Use thread safe collections when needed.

Collection Best Practices.txt

Core Java

Java I/O & File Handling

Lesson Objectives

- To understand the following topics:
  - Streams
  - Byte Stream I/O hierarchy
  - Character Streams: Readers and Writers
  - The File Class
  - Object Serialization
  - Scanning and Formatting

Overview of I/O Streams

- Most programs need to access external data.
- Data is retrieved from an input source.
  - Program results are sent to output destination.

Figure 8-1: A program uses an input stream to read data from a source, one item at a time

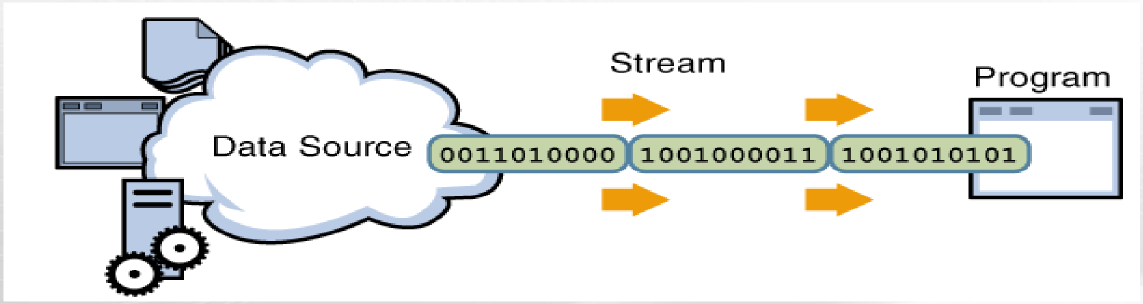
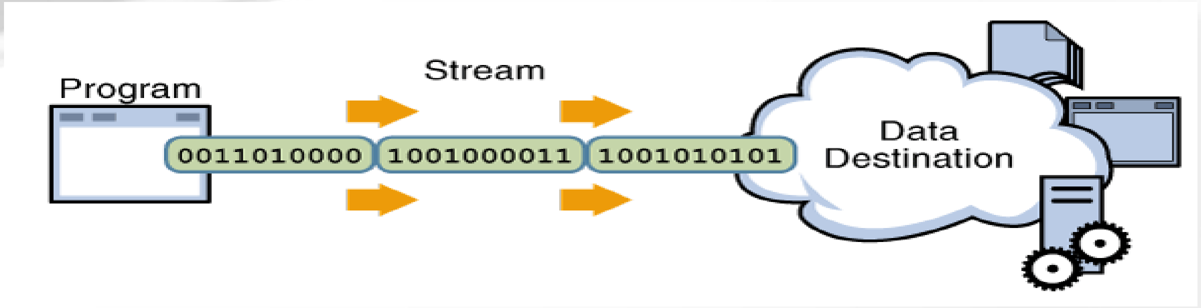


Figure 8-2:A program uses an output stream to write data to a destination, one item at time



What is a Stream?

- Abstraction that consumes or produces information.
- Linked to source and destination.
- Java implements streams within class hierarchies defined in the *java.io* package.
- An *input* stream acts as a *source* of data.
- An *output* stream acts as a *destination* of data.

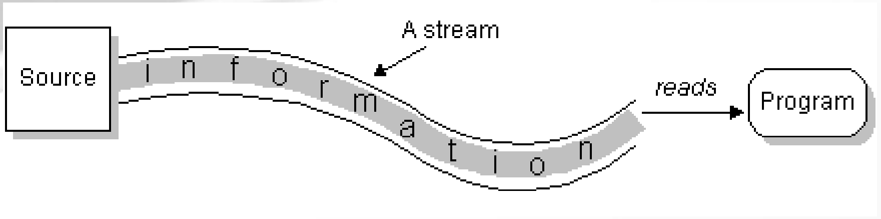


Figure 8-3: (a) Input Stream

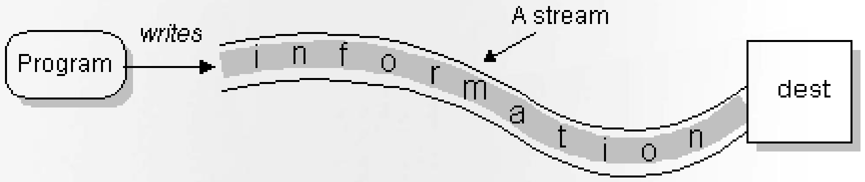
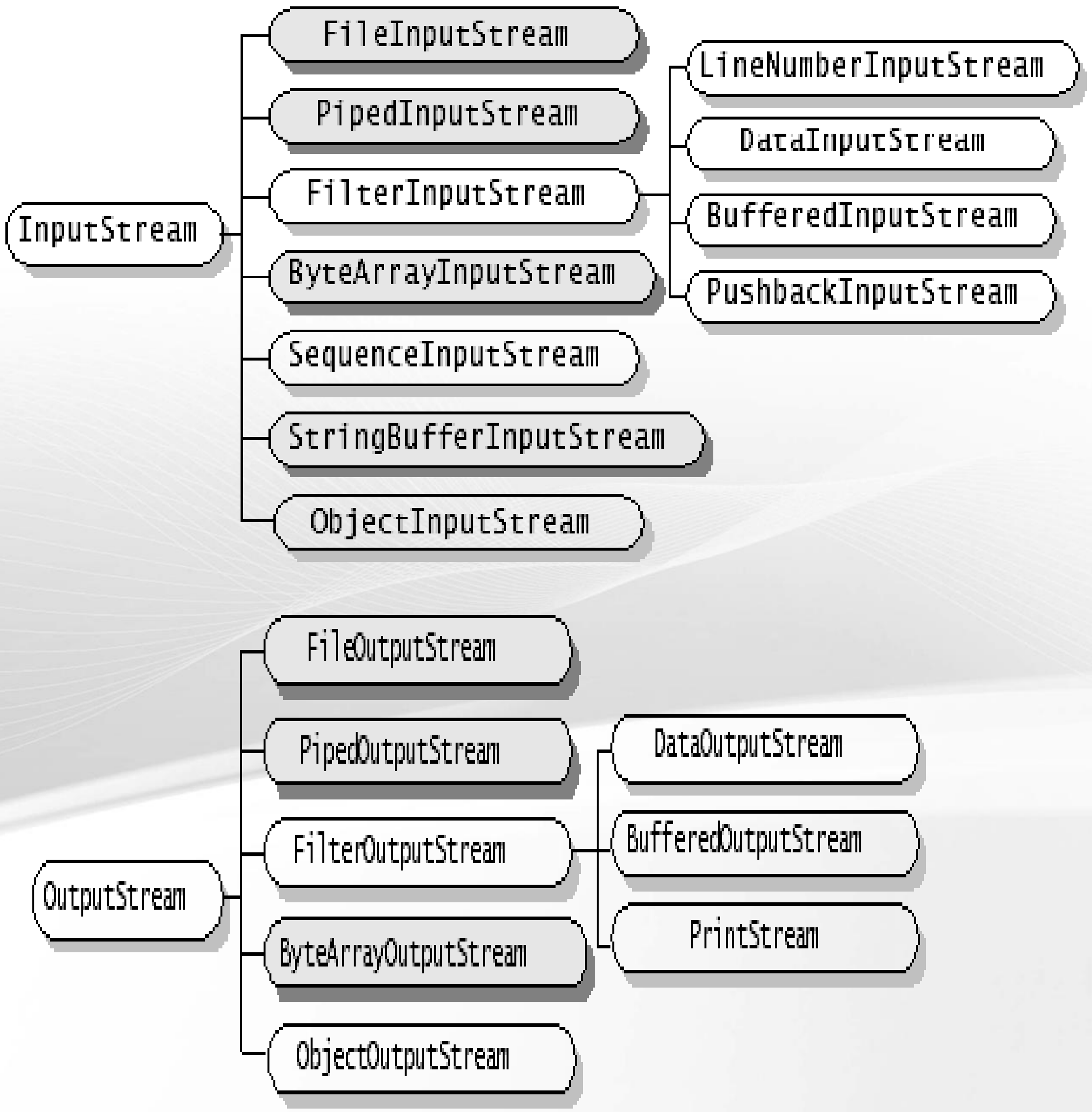


Figure 8-3:(b) Output stream

Types of Streams

- *Byte Streams*: Handle I/O of raw binary data.
- *Character Streams*: Handle I/O of character data. Automatic translation handling to and from a local character.
- *Buffered Streams*: Optimize input and output with reduced number of calls to the native API.
- *Data Streams*: Handle binary I/O of primitive data type and String values.
- *Object Streams*: Handle binary I/O of objects.
- *Scanning and Formatting*: Allows a program to read and write formatted text.

Byte Stream I/O Hierarchy



InputStream Class - Methods

Method	Description
close()	Closes this input stream and releases any system resources associated with the stream.
int read()	Reads the next byte of data from the input stream.
int read(byte[] b)	Reads some number of bytes from the input stream and stores them into the buffer array *b*.
int read(byte[] b, int off, int len)	Reads up to *len* bytes of data from the input stream into an array of bytes.

OutputStream Class - Methods

Method	**Description**
close()	Closes this output stream and releases any system resources associated with this stream.
flush()	Flushes this output stream and forces any buffered output bytes to be written out.
write(byte[] b)	Writes *b.length* bytes from the specified byte array to this output stream.
write(byte[] b, int off, int len)	Writes *len* bytes from the specified byte array starting at offset off to this output stream.
write(int b)	Writes the specified byte to this output stream.

InputStream Subclasses

Classname	Description
DataInputStream	A filter that allows the binary representation of java primitive values to be read from an underlying inputstream
BufferedInputStream	A filter that buffers the bytes read from an underlying input stream. The buffer size can be specified optionally.
FilterInputStream	Superclass of all input stream filters. An input filter must be chained to an underlying inputstream.
ByteArrayInputStream	Data is read from a byte array that must be specified
FileInputStream	Data is read as bytes from a file. The file acting as the input stream can be specified by File object, or as a String
PushBackInputStream	A filter that allows bytes to be “unread “ from an underlying stream. The number of bytes to be unread can be optionally specified.
ObjectInputStream	Allows binary representation of java objects and java primitives to be read from a specified inputstream.
PipedInputStream	It reads many bytes from PipedOutputStream to which it must be connected.
SequenceInputStream	Allows bytes to be read sequentially from two or more input streams consecutively.

FileInputStream and FileOutputStream Example

```
import java.io.*;
class CopyFile  {
                FileInputStream fromFile;
                FileOutputStream toFile;
     public void init(String arg1, String arg2) {
                //Assign the files
     try{
                fromFile = new FileInputStream(arg1);
                toFile = new FileOutputStream(arg2);
     } catch (FileNotFoundException fnfe) {
                System.out.println("Exception: " + fnfe);
     } catch (IOException ioe) {
                System.out.println("Exception: " + ioe);
     } catch (ArrayIndexOutOfBoundsException aioe) {
                  System.out.println("Exception: " + aioe);
     }
 }
```

FileInputStream and FileOutputStream Example

```
public void copyContents() {// copy bytes
    try {
            int i = fromFile.read();
            while ( i != -1)
              { //check the end of file
                toFile.write(i);
                i = fromFile.read();
               }
              } catch (IOException ioe)
           { System.out.println("Exception: " + ioe);}
 }
```

FileInputStream and FileOutputStream Example

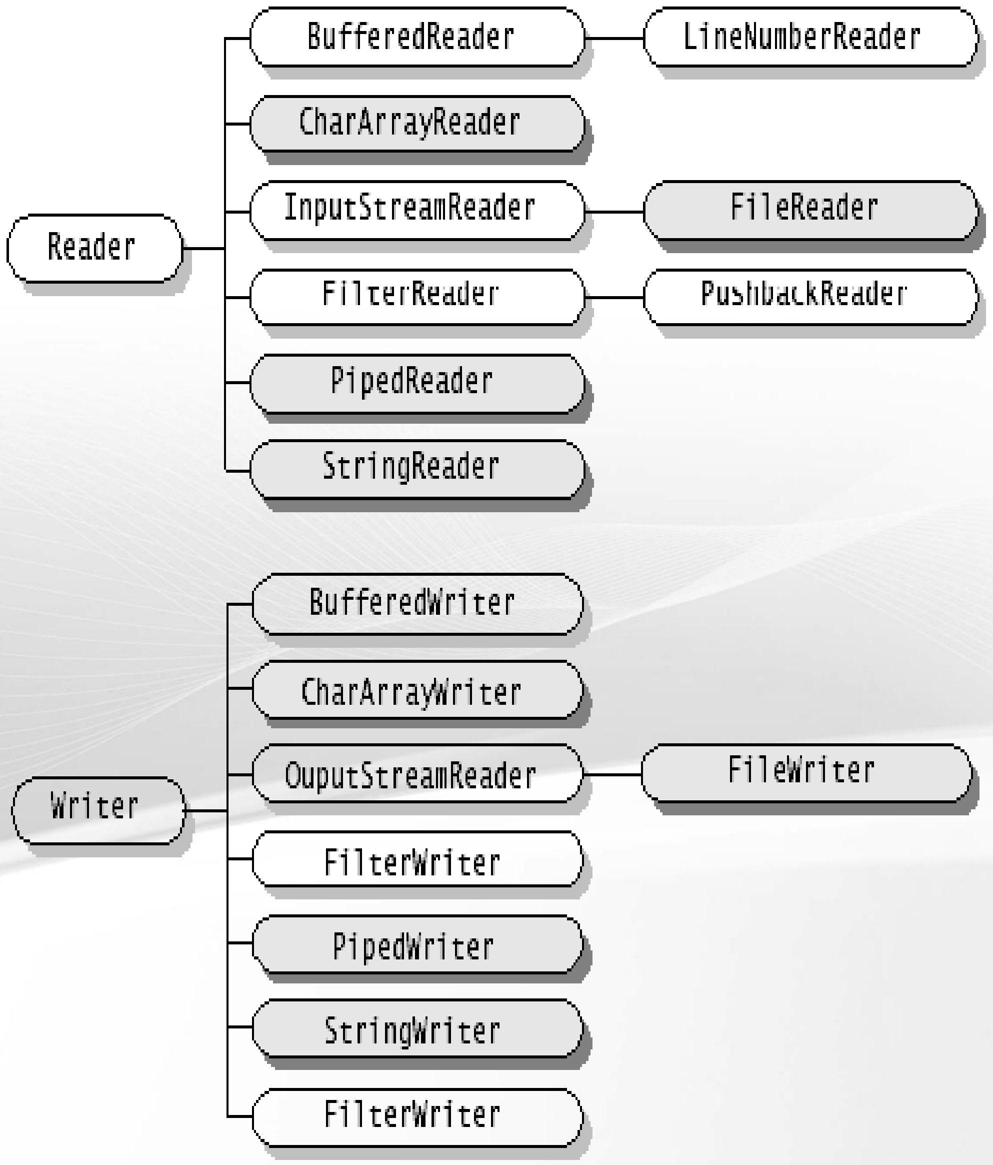
```
public void closeFiles() {//close the files
                                try{
          fromFile.close();
          toFile.close();
          } catch (IOException ioe)
                 { System.out.println("Exception: " + ioe);
         }
    }

public static void main(String[] args){
         CopyFile c1 = new CopyFile();
         c1.init(args[0], args[1]);
         c1.copyContents();
         c1.closeFiles(); } }
```

Demo : Input/Output Streams

Demo:

CopyFile.java

Character Stream I/O Hierarchy



Character Streams Classes - Readers and Writers

- All Character Stream classes are descended from *Reader* and *Writer* Abstract classes.
- Java represents characters internally in the 16-bit Unicode character encoding.
- *Reader* is an input character stream that reads a sequence of Unicode characters.
- *Writer* is an output character stream that writes a sequence of Unicode characters

Reader Class - Methods

Method	**Description**
int read() throws IOException	reads a byte and returns as an int
int read(char b[])throws IOException	reads into an array of chars b
int read(char b[], int off, int len) throws IOException	reads *len* number of characters into char array *b*, starting from offset *off*
long skip(long n) throws IOException	Can skip n characters.

Writer Class - Methods

Method	**Description**
void write(int c)throws IOException	writes a byte.
void write(char b[])throws IOException	writes from an array of chars b
void write(char b[], int off, int len) throws IOException	writes *len* number of characters from char array *b*, starting from offset *off*
void write(String b, int off, int len) throws IOException	writes *len* number of characters from string *b*, starting from offset *off*

FileReader and FileWriter Example

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class CopyCharacters {
    public static void main(String[] args) throws IOException {
        FileReader inputStream = null;
        FileWriter outputStream = null;
        try {
            inputStream = new FileReader(“sampleinput.txt");
            outputStream = new FileWriter(“sampleoutput.txt");
            int c;
            while ((c = inputStream.read()) != -1) {
                outputStream.write(c);
            }
```

FileReader and FileWriter Example

```
} finally {
   if (inputStream != null) {
 inputStream.close();
 }
    if (outputStream != null)
 {

outputStream.close();
 }

 }
 } }
```

Buffered Input Output Stream

- An unbuffered I/O means each read or write request is handled directly by the underlying OS.
  - Makes a program less efficient.
    - Each such request often triggers disk access, network activity, or some other relatively expensive operation.
  - Java implements buffered I/O Streams to reduce this overhead.
    - 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.

Buffered Stream

- A program can convert a un-buffered stream into buffered using the *wrapping idiom:*
  - Un-buffered stream object is passed to the constructor of a buffered stream class.
  - Example

```
inputStream =
new BufferedReader(new FileReader(“sampleinput.txt"));

outputStream =
new BufferedWriter(new FileWriter(“sampleoutput.txt"));
```

Demo : Buffered Stream

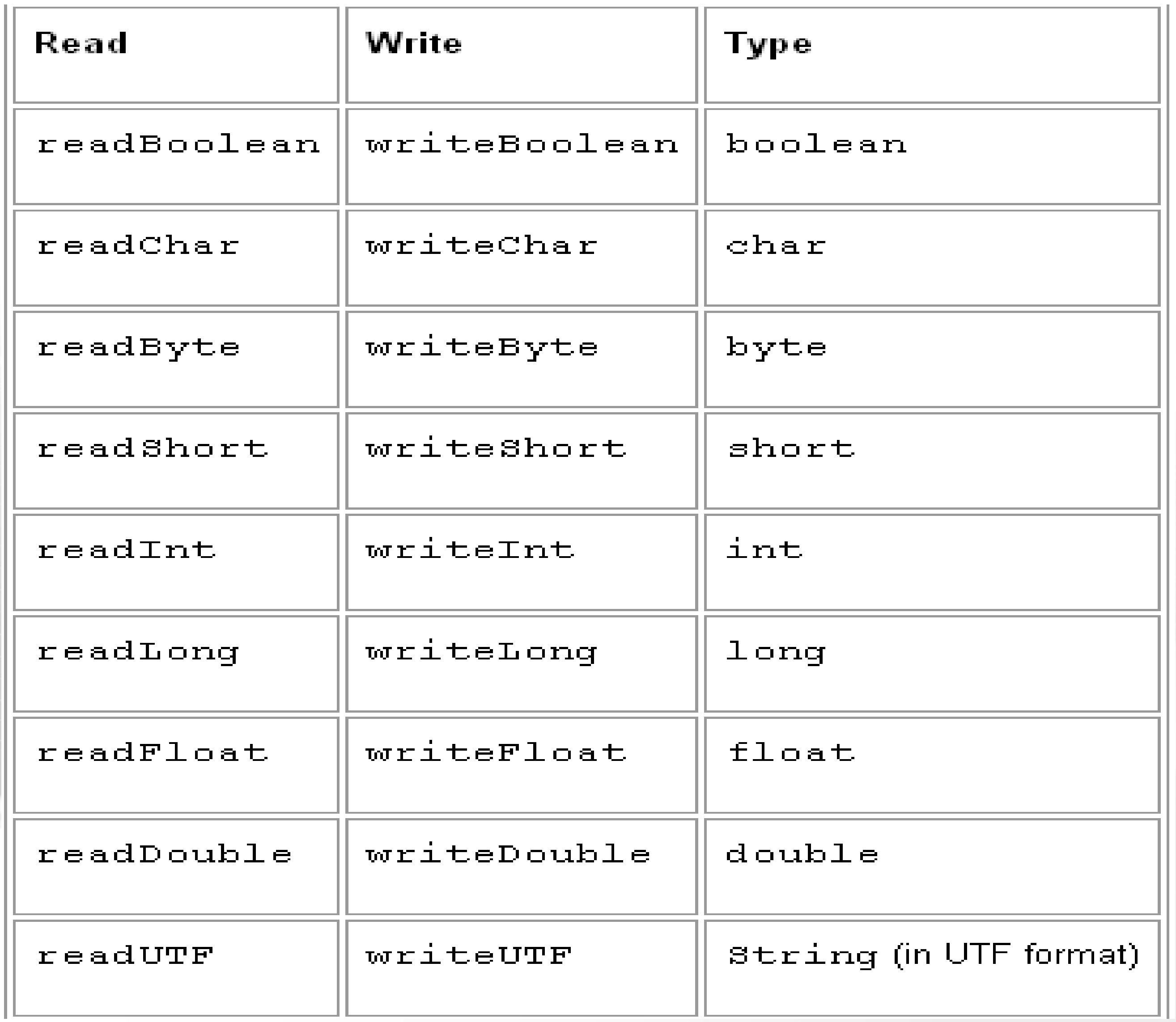
- **Demo:**

  CharEncode.java

DataInputStream / DataOutputStream Classes

- Data Streams:
  - Support binary I/O of primitive data type values:
    - boolean, char, byte, short, int, long, float, and double as well as String.
  - All data streams implement either the *DataInput* interface or the *DataOutput* interface.
    - DataInputStream and DataOutputStream are most widely-used implementations.

DataInputStream / DataOutputStream Methods



Read	Write	Type
readBoolean	writeBoolean	boolean
readChar	writeChar	char
readByte	writeByte	byte
readShort	writeShort	short
readInt	writeInt	int
readLong	writeLong	long
readFloat	writeFloat	float
readDouble	writeDouble	double
readUTF	writeUTF	String (in UTF format)

DataInputStream / DataOutputStream Classes Demo

```
public static void writeData(double[] data, String file)throws
     IOException {
    OutputStream fout = new FileOutputStream(file);
    DataOutputStream out = new DataOutputStream(fout);
    out.writeInt(data.length);
    for (double d : data) {     out.writeDouble(d); }
    out.close();
 }
public static double[] readData(String file) throws IOException {
    InputStream fin = new FileInputStream(file);
    DataInputStream in = new DataInputStream(fin);
    double[] data = new double[in.readInt()];
    for (int i = 0; i < data.length; i++) {      data[i] = in.readDouble(); }
    in.close();
    return data;
 }
```

DataInputStream / DataOutputStream Classes Demo

- **Demo:**

JavaPrimitiveValues.java

STDIO.java

ObjectInputStream / ObjectOutputStream

- Object streams support I/O of objects:
  - Support I/O of primitive data types.
  - Object has to be *Serializable* type.
  - *Object Classes*: ObjectInputStream, ObjectOutputStream
    - Implement ObjectInput and ObjectOutput, which are sub interfaces of DataInput and DataOutput.
  - An object stream can contain a mixture of primitive and object values.

Object Serialization

- *Object Serialization:*
  - Object Serialization allows an object to be transformed into a sequence of bytes that can be later re-created (deserialized) into an original object.
  - Process to read and write objects.
  - Provides ability to read or write a whole object to and from a raw byte stream.
  - Use object serialization in the following ways:
    - *Remote Method Invocation (RMI):* Communication between objects via sockets.
    - *Lightweight persistence:* Archival of an object for use in a later invocation of the same program.

Object Serialization - Example

```
import java.io.*;
// This is a serializable object
class Employee implements Serializable  {
        String name;
        int age;
        double salary;
        Employee(String name, int age, double salary) {
                this.name = name;
                this.age = age;
                this.salary = salary;
        }
        public void showDetails() {
                System.out.println("Name        :" + name);
                System.out.println("Age         :" + age);
                System.out.println("Salary      :" + salary);
        }
}
```

Object Serialization - Example

```
class ObjectSerializationDemo {
     void writeData() {
       Employee db[] = {
                          new Employee("Sachin",25,12000.56),
                          new Employee("Rahul",24,12670.78),
                          new Employee("Hritik",28,16000.89)
                          };
      try  {
           FileOutputStream out = new FileOutputStream("emp-obj.dat");
           ObjectOutputStream sout = new ObjectOutputStream(out);
                          for (int i = 0; i < db.length ; i++ ) {
                                     sout.writeObject(db[i]);
                          }
                          sout.close();
                          } catch (IOException ioe) {
                                     ioe.printStackTrace();
```

Object Serialization - Example

```
        }
            }
            void readData() {
try         {
                    FileInputStream in = new FileInputStream("emp-obj.dat");
                    ObjectInputStream sin = new ObjectInputStream(in);
                    Employee e = (Employee) sin.readObject();
                    e.showDetails();
                    e = (Employee) sin.readObject();
                    e.showDetails();
                    e = (Employee) sin.readObject();
                    e.showDetails();
                    sin.close();
```

Object Serialization - Example

```
                } catch (IOException ioe) {
                                        ioe.printStackTrace();
                } catch (ClassNotFoundException cnfe) {
                                        cnfe.printStackTrace();
          }
             }
             public static void main(String  args[]) {
                 ObjectSerializationDemo impl = new ObjectSerializationDemo();
                        impl.writeData();
                        impl.readData();
             }
        }
```

Demo : Object Serialization

- **Demo:**

ObjectSerializationDemo.java

Scanning and Formatting

- Java platform provides two APIs to translate to and from neatly formatted data.
  - Scanner API
    - Breaks input into individual tokens associated with bits of data.
  - Formatting API
    - Assembles data into nicely formatted, human-readable form.

Scanner Class

- Prior to Java 1.5 getting input from the console involved multiple steps.
- Java 1.5 introduced the *Scanner* class to simplify console input.
- Also reads from files and Strings (among other sources).
- Used for powerful pattern matching.
- Scanner is in the Java.util package.
  - Hence, type the following code:

```
import java.util.Scanner;
```

Scanner Class

- Scanner(File source):
  - Constructs a new Scanner that produces values scanned from the specified file.
- Scanner(InputStream source):
  - Constructs a new Scanner that produces values scanned from the specified input stream.
- Scanner(Readable source):
  - Constructs a new Scanner that produces values scanned from the specified source.
- Scanner(String source):
  - Constructs a new Scanner that produces values scanned from the specified string.

How to use Scanner class?

- Scanner class basically parses input from the source into tokens by using delimiters to identify the token boundaries.
- The default delimiter is whitespace.
- Example:

```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt();
System.out.println("You entered" + i);
```

Demo : Scanner class

- **Demo:**

ParseString.java

Scanner Methods

- String next()
- boolean nextBoolean()
- byte nextByte()
- double nextDouble()
- float nextFloat()
- int nextInt()
- String nextLine()
- long nextLong()
- short nextShort()

- boolean hasNext()
- boolean hasNextBoolean()
- boolean hasNextByte()
- boolean hasNextDouble()
- boolean hasNextFloat()
- boolean hasNextInt()
- boolean hasNextLong()
- boolean hasNextShort()

Format Method

Formats multiple arguments based on a format string.
Example:

```
public class Root2 {
   public static void main(String[] args) {
              int i = 2;
 double r = Math.sqrt(i);
System.out.format("The square root of %d is %f.%n", i, r);
}

}
```

Output: The square root of 2 is 1.414214.