Chapter 1
Functional Programming

Functional programming

- Functional programming is a paradigm where the basic unit of computation is a function.
- A style of programming that treats computation as the evaluation of mathematical functions
- Here functions are not the methods we write in programming.
- Methods are procedures where we write a sequence of instructions to tell the computer what to do.
- In functional programming, functions are treated like mathematical functions where we **map inputs to outputs to produce a result**. Eg. f(x)=3x.

Functional programming …

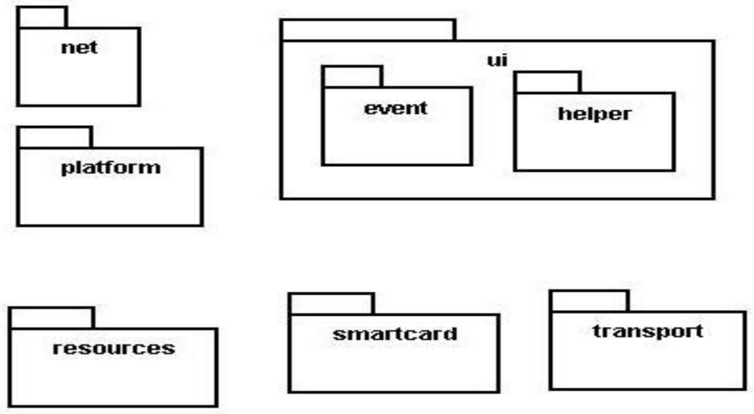
- In functional programming, the software is developed around the evaluation of functions.
- The functions are isolated and independent, and they **depend only on the arguments passed to them and do not alter the program's state like modifying a global variable**.
- Treats data as being immutable
- Functions can take functions as arguments and return functions as results

Functional programming

- Key characteristics of functional programming include:
  - **Pure Functions:** Functions that produce the same output for a given input and have no side effects. Pure functions do not modify state or rely on external data.
  - **Immutable Data:** Data that cannot be changed after it is created. Instead of modifying existing data, functional programming encourages creating new data structures with modifications.
  - **Declarative Programming:** We will discuss about this in the future

Java Packages

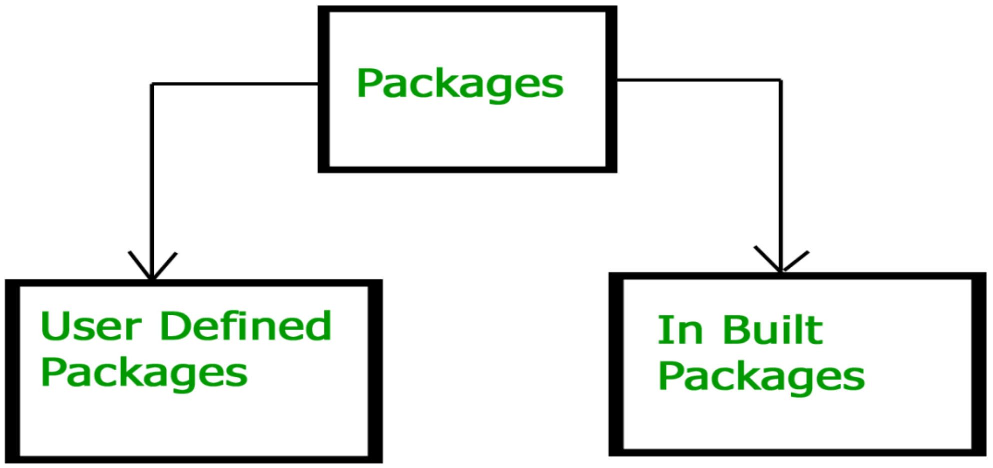
- Java contains many predefined classes that are grouped into categories of related classes called packages.
- **Package**: A collection of related classes.
  - Can also "contain" sub-packages.
  - Packages are Java's way of doing large-scale design and organization.
  - They are used both to categorize and group classes.



Java Packages

- Uses of Java packages:
  - Group related classes together as a *namespace* to avoid name collisions
  - Provide a layer of access / protection
  - Keep pieces of a project down to a manageable size
- Preventing naming conflicts.
  - For example there can be two classes with name Employee in two packages, college.staff.se.Employee and college.staff.ee.Employee
  - import java.util.Date;
  - import my.package.Date;
- Packages can be considered as data encapsulation (or data-hiding).
- Packages naming convention is in reverse order of domain names, i.e., et.edu.se.ceme.aastu

Package Types



Java System Packages & Their Classes

- Java.lang
  - Language support classes
  - These are classes that java compiler itself uses & therefore they are automatically imported
  - They include classes for primitive types, strings, maths function, threads & exception

Package Types…

Java System Packages & Their Classes

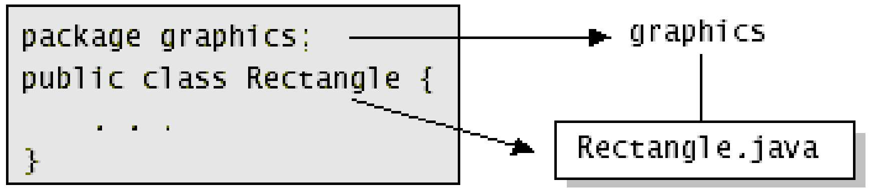
- Java.util
  - Language Utility classes such as vector, hash tables, random numbers, date etc
- Jave.IO
  - Input/Output support classes
  - They provide facilities for the input & output of data
- Java.awt
  - Set of classes for implementing graphical user interface
  - They include classes for windows, buttons, list, menus & so on
- Java.net
  - Classes for networking
  - They include classes for communicating with local computers as well as with internet servers
- Java.applet
  - Classes for creating and implementing applets

Packages and directories

- package ←→ directory (folder)
- class ←→ file

- A class named D in package a.b.c should reside in this file:

  a/b/c/D.class

```
package graphics;
public class Rectangle {
    . . .
}
```

  - (relative to the root of your project)

A package declaration

```
package name;

public class name { ...


Example:
package pacman.model;

public class Ghost extends Sprite {
    ...
}
```

- File Sprite.java should go in folder pacman/model .

Importing a package

```
import packageName.*;                    // all classes
```

Example:

```
package pacman.gui;
import pacman.model.*;

public class PacManGui {
    ...
    Ghost blinky = new Ghost();
}
```

- PacManGui must import the model package in order to use it.

Importing a class & Referring to packages

import **packageName.className;** // one class

- Importing single classes has high precedence:
  - if you import .*, a same-named class in the current dir will override

- Referring to packages

packageName.className

Example:

```
java.util.Scanner console =
    new java.util.Scanner(java.lang.System.in);
```

The default package

- Compilation units (files) that do not declare a package are put into a default, unnamed, package.
- Classes in the default package:
  - Cannot be imported
  - Cannot be used by classes in other packages
- Many editors discourage the use of the default package.
- Package java.lang is implicitly imported in all programs by default.
  - import java.lang.*;

Package access

- Java provides the following access modifiers:
  - public : Visible to all other classes.
  - private : Visible only to the current class (and any nested types).
  - protected : Visible to the current class, any of its subclasses, and any other types within the same package.
  - default (package): Visible to the current class and any other types within the same package.
- To give a member default scope, do not write a modifier:

```
package pacman.model;
public class Sprite {
    int points;      // visible to pacman.model.*
    String name;     // visible to pacman.model.*
```

Collections

2. Introduction

- Java collections framework
  - prebuilt data structures
  - interfaces and methods for manipulating those data structures

Collections Overview

- ★ A collection is a data structure—actually, an object—that can hold references to other objects.
  - – Usually, collections contain references to objects that are all of the same type.
- ★ Is simply an object that groups multiple elements into a single unit.
- ★ They are used to store, retrieve, manipulate, and communicate aggregate data.

What Is a Collections Framework?

- ★ A collections framework is a unified

Collections…

- All collections frameworks contain the following:
- Interfaces: are Abstract Data Types that represent collections.
  - Interfaces allow collections to be manipulated independently of the details of their representation.
  - In object-oriented languages, interfaces generally form a hierarchy.
- Implementations: are the concrete implementations of the collection interfaces.
  - In essence, they are reusable data structures.

Collections…

Benefits of the Java Collections Framework

- The Java Collections Framework provides the following benefits:
- **Reduces programming effort:**
  - By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.
  - By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:**
  - This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
  - The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations.

Collections…

Benefits of the Java Collections Framework

- **Allows interoperability among unrelated APIs:**
  - The collection interfaces are the vernacular by which APIs pass collections back and forth.
  - If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:**
  - Many APIs naturally take collections on input and furnish them as output.
  - In the past, each such API had a small sub-API devoted to manipulating its collections.
  - There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them.
  - With the advent of standard collection interfaces, the problem went away.

Collections…

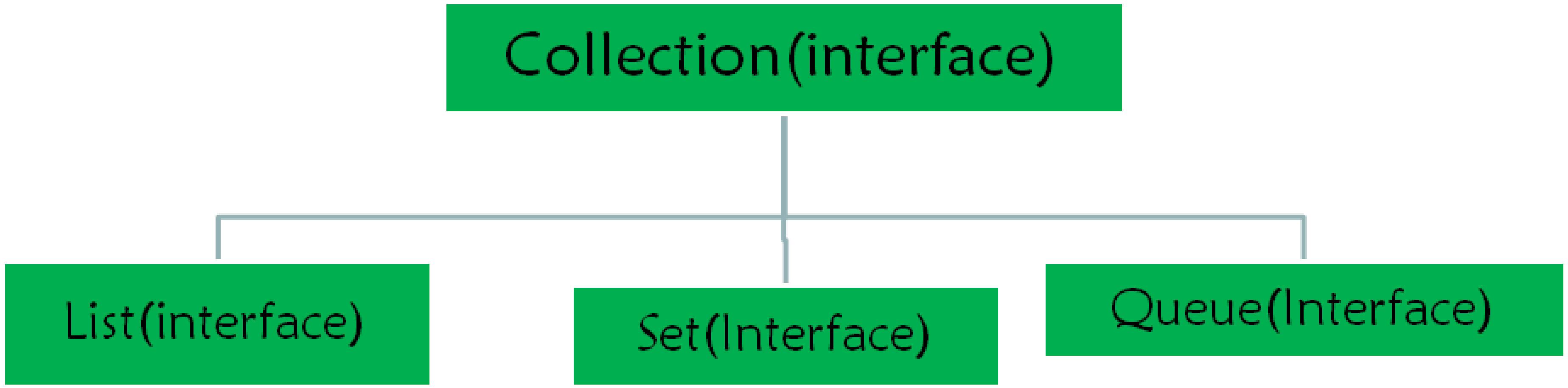
Benefits of the Java Collections Framework

- **Reduces effort to design new APIs:**
  - This is the flip side of the previous advantage.
  - Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:**
  - New data structures that conform to the standard collection interfaces are by nature reusable.
  - The same goes for new algorithms that operate on objects that implement these interfaces.

Collections…

- Collections
  - A class from java.util package or it is a predefined class in java.util
  - Contains certain methods that can be used for other collection objects
  - Allow to perform operations in collection object

  E.g for ArrayList (it is a collection object)
  - We can use collections.sort(), collections.shuffle()…



Type-Wrapper Classes for Primitive Types

- What is type-wrapper?
- Each primitive type has a corresponding type-wrapper class (in package `java.lang`).
  - Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- Each type-wrapper class enables you to manipulate primitive-type values as objects.
- Collections cannot manipulate variables of primitive types.
  - They can manipulate objects of the type-wrapper classes, because every class ultimately derives

Type-Wrapper Classes for Primitive Types (cont.)

- Each of the numeric type-wrapper classes—`Byte`, `Short`, `Integer`, `Long`, `Float` and `Double`—extends class `Number`.
- The type-wrapper classes are `final` classes, so you cannot extend them.
- Primitive types do not have methods, so the methods related to a primitive type are located in the corresponding type-wrapper class.

Autoboxing and Auto-Unboxing

- A boxing conversion converts a value of a primitive type to an object of the corresponding type-wrapper class.
- An unboxing conversion converts an object of a type-wrapper class to a value of the corresponding primitive type.
- These conversions can be performed automatically (called autoboxing and auto-unboxing).
- Example:

  -
    ```
    // create integerArray
    Integer[] integerArray = new Integer[ 5 ];

    // assign Integer 10 to integerArray[ 0 ]
    integerArray[ 0 ] = 10;

    // get int value of Integer int value =
    integerArray[ 0 ];
    ```

Autoboxing & Auto-Unboxing …

- In this case, autoboxing occurs when assigning an int value (10) to integerArray[0], because integerArray stores references to Integer objects, not int values.
- Auto-unboxing occurs when assigning integerArray[0] to int variable value, because variable value stores an int value, not a reference to an Integer object.
- Boxing conversions also occur in conditions, which can evaluate to primitive boolean values

Interface Collection and Class Collections

- Interface Collection is the root interface from which interfaces `Set`, `Queue` and `List` are derived.
- Interface Set defines a collection that does not contain duplicates.
- Interface Queue defines a collection that represents a waiting line.
- Interface `Collection` contains bulk operations for adding, clearing and comparing objects in a collection.
- A `Collection` can be converted to an array.
- Interface `Collection` provides a method that returns an Iterator object, which allows a program to walk through the collection and remove elements from the collection during the iteration.
- Class Collections provides `static` methods that search, sort and perform other operations on collections.

Lists

- A `List` (sometimes called a sequence) is a `Collection` that can contain duplicate elements.
- `List` indices are zero based.
- In addition to the methods inherited from `Collection`, `List` provides methods for manipulating elements via their indices, manipulating a specified range of elements, searching for elements and obtaining a ListIterator to access the elements.
- Interface `List` is implemented by several classes, including ArrayList, LinkedList and Vector.
- Autoboxing occurs when you add primitive-type values to objects of these classes, because they store only references to objects.

ArrayList and Iterator

- List method add adds an item to the end of a list.
- List method size return the number of elements.
- List method get retrieves an individual element's value from the specified index.
- Collection method iterator gets an `Iterator` for a `Collection`.
- Iterator- method hasNext determines whether a `Collection` contains more elements.
  - Returns `true` if another element exists and `false` otherwise.
- Iterator method next obtains a reference to the next element.
- `Collection` method contains determine whether a `Collection` contains a specified element.
- Iterator method remove removes the current element from a `Collection`.

Example

```
// CollectionTest.java
// Using the Collection interface.
import java.util.List;
import java.util.ArrayList; import java.util.Collection;
import java.util.Iterator;

public class CollectionTest
{ private static final String[] colors =
{ "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
private static final String[] removeColors = { "RED",
"WHITE", "BLUE" };

// create ArrayList, add Colors to it and manipulate it
```

Example

```
public CollectionTest()
{
List< String > list = new ArrayList< String >();
List< String > removeList = new ArrayList< String >();

// add elements in colors array to list
for ( String color : colors )
list.add( color );

// add elements in removeColors to removeList
for ( String color : removeColors )
removeList.add( color );

System.out.println( "ArrayList: " );

```

Example

```
// output list contents
for ( int count = 0; count < list.size(); count++ )
System.out.printf( "%s ", list.get( count ) );

// remove colors contained in removeList
removeColors( list, removeList );

System.out.println( "\nArrayList after calling removeColors: " );

// output list contents
for ( String color : list )
System.out.printf( "%s ", color );
} // end CollectionTest constructor

// remove colors specified in collection2 from collection1
```

Example

```
private void removeColors(
Collection< String > collection1, Collection< String > collection2 )
{
// get iterator
Iterator< String > iterator = collection1.iterator();

// loop while collection has items
while ( iterator.hasNext() )

if ( collection2.contains( iterator.next() ) )
iterator.remove(); // remove current Color
} // end method removeColors
```

Example

```

public static void main( String args[] )
{
new CollectionTest();
} // end main
} // end class CollectionTest

```

Lambda Expression

- Output: [1, 2, 3, 4, 5]
- In this example, we created an anonymous inner class i.e. new Comparator<Integer>() { } that implements the Comparator interface and overrides the compare() method.
- Lambda Expression
- *Lambda expression is an anonymous function that takes in parameters and returns a value*. It is called an anonymous function because it doesn't require a name.
- Syntax: (parameter1, parameter2, ...) -> expression

  (parameter1, parameter2, ...) -> { body }

Java Lambda Expression Syntax

- The Syntax of Lambda Expression in Java consists of three components.
- **Arguments-list**: It is the first portion of the Lambda expression in Java. It can also be empty or non-empty. In the expression, it is enclosed by a round bracket.
- **Lambda operator ->:** It's an arrow sign that appears after the list of arguments. It connects the arguments-list with the body of the expression.
- **Body**: It contains the function body of lambda

Lambda Expression …

```
import java.util.ArrayList;  import java.util.Arrays;
import java.util.List;
public class Main {
  public static void main(String[] args) {
   List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5));
   int sum = list.stream().reduce(0, (a, b) -> a + b);
   System.out.println(sum);
 }
}
```

- **Output: 15**

Lambda Expression

```
interface I
      {
           public void method1( ); // functional interface
           default void method2();
                {
                }
      }
    Functional Interface Annotation
  Example
    package demos
    @ functional interface   // annotation optional
    interface Library
    {
        public void bookLibrary();
    }
```

Lambda Expression

```
class java implements Library
    {
        public void bookLibrary()
         {
           System.out,println(“ Java Complete reference is a book”);
         }
   }
 public class Test
     {
       public static void main(String[] args)
         {
               java Library = new java ( );
               Library.bookLibrary( );
         }
     }
}
```

Lambda Expression

```
package demos
@ Functional interface
 interface Library
  {
   public void bookLibrary();
  }
 public class Test
  {
   public static void main(String[] args)
   {
       Library p= ( )->System.out.println(“JCR book is booked….”);
         p.bookLibrary();
   }
  }
```

Lambda Expression

Example

```
package demos;
 interface Library
   {    public void bookLibrary(String source, String destination);    }
 class Java implements Library
  {
    public void bookLibrary (String source, String destiontion)
     {
       System.out.println("JCR a is book from " + source + " To "
           +destination);
      }
    }
  pbulic static void main(String[ ] args)
  {
      Library a=new bookLibrary ();
      a. bookLibrary("Addiss Ababa", "Dire Dawa");
  }
```

Lambda Expression

Lambda expression equivalent of the above code

```
package demos;
 interface Library
   {
      public void bookLibrary(String source, String destination);
   }

      public class Test
     {
       public static void main(String[ ] args)
       {
         Library a=  (Source, destination)-> System.out.println(“ Ola cab ia
         booked from  ” + source + “To ”+ destination);
         a. bookLibrary(“Addiss Ababa”, “Dire Dawa”);
        }
   }
```

Lambda Expression

Without functional Interface we can use Lambda expression

Predefined Functional Interface – using these interface we can write Lambda ex

Java predefined Functional Interfaces

Predicate, Function , Consumer , Supplier

Java.util.function.*; – package contain all these predefined functional interface

Predicate Interface

- Contains a single Abstract Method known as test( )
- Prototype of predicate Interface

```
 interface predicate(T)
{
    public abstact boolean test( ); // takes always one
     //argument of any type (float, int, String) and it return
     //boolean value
}
```

When we do conditional statements, we use predicate functional interface

Lambda Expression

How to use predicate Interface along with test method to invoke lambda expressions

Example

```
   interface predicate <T>
   {
     public abstract boolean test (T t)
   }
Package  predicates;
Import java.util.function.Predicate;
  //predicate ---- > one parameter returns boolean
 //use only if you have conditional checks in your program
public class Demo1
{
     public static void main(String [ ] args)
     {
             //Ex1
           predicate<Integer> p = i→ (i>10);
           System.out.println(p.test(20)); // true
```

Lambda Expression

```
//Ex1
        predicate<Integer> p = i->(i>10);
        System.out.println(p.test(20)); // true
        System.out.println(p.test(5); // false
//Ex2 : check the length of given string is greater than 4 or not
        predicate<String> pr=s->(s.length()>4);
        System.out.println(pr.test("Welcome")); // true
        System.out.println(pr.test("abc")); // false
//Ex3: print array elements whose size is > from array
       String name [ ] ={" David ", "Scott", "Smith", "John", "Marry"};
       for(String names:name)
          {
             if(pr.test(names))
               {    System.out.println(name);     }
           }
       }// - we can specify multiple conditions in lambda expression
  }
```

Lambda Expression

Example Demo2

```
package predicates;
Import java.util.function.Predicate;
class Employee
 {
      String ename;
      int Salary;
      int experience;
      Employee(String name, int sal, int exp)
        {
           ename=name;    Salary=sal;    experience= exp
        }
 }
   public class Demo2
    {
         public static void  main(String [ ] args)
          { //Example 1
               Employee emp =new Employee(“John” , 5000, 5);
```

Lambda Expression

```
  { //Example 1
            Employee emp =new Employee(“John” , 5000, 5);
   // Emp obj -> return name and salary if sal>30k and exp >3
 Predicate <Employee> pr = e->(e.salary>3000 && e.experience >3);
        System.out.println(pr.test(emp)); //display true
   //Example2, using array List to store a number of employee objects
       ArrayList<Employee> al =new ArrayList <Employee> ( );
         al.add(new Emloyee (“John”, 50000, 5));
         al.add(new Emloyee (“David”, 20000, 2));
         al.add(new Emloyee (“Scott”, 30000, 3));
         al.add(new Emloyee (“Mery”, 40000, 6));
          for(Employee   e:al)
             {
               if(pr.test(e))
                 {
                   System.out.println(e.name  + “     ” + e.salary);
                  }
             }        } } }
```

Lambda Expression

Example Demo3

Join multiple predicate each have one lambda expression

```
Package predicate;
  import java.util.function.Predicate;
      // joining predicate by and, or and negate
     // p1 ----- checks number is even
     // p2 ----- checks greater than 50
   public class Demo3
   {
     public static void main(String [ ] args)
       {
        int a[ ]= {5, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65};
        Predicate<Integer> p1 = i -> i%2==0;
        Predicate<Integer> p2 = i -> i>50;
```

Lambda Expression

```
// and
     System.out.println(“ Following are numbers Even & Greater than 50”);
     for( int n: a)
     {     if(p1.and(p2).test(n)) // if(p1. test(n) && p2.test(n))
                {    System.out.println(n); }
     }
//or
 for( int n: a)
     {     if(p1.or(p2).test(n)) // if(p1. test(n) && p2.test(n))
           { System.out.println(n); // 20 30 40 50 55 60 65 }
     }
//negate
for( int n: a)
     {    if(p1.negate( ).test(n)) // if(p2.negate( ).test(n))
           { System.out.println(n);}
     }
   }
}
```

Lambda Expression

Predefined Functional Interface –Function

Function

- It has only Abstract method known as apply( )
- It can take any type of parameter and return a single value of any type

```
// prototype of this interface
 interface Function <T, R> // T for the argument, R return value
    {
     public R apply(T);
    }
```

Used to do some kind of operation

How to work with Functional Interface?

Lambda Expression

Predefined Functional Interface –Function

Example Demo1

```
    package functions;
    import java.util.function.Function;
    public class Demos1
    {
      public static void main(String [ ] args)
      {
        Function <Integer, Integer> f=n ->n*n;
        System.out.println(f.apply(5)); // 25
        System.out.println(f.apply(10)); // 100
        System.out.println(f.apply(2)); // 4
//String length
        Function <String, Integer> fn = s->s.length();
        System.out.println(fn.apply(“Welvome”)); // 7
        System.out.println(fn.apply(“Java Programming”)); // 16
   }
}
```

Lambda Expression

Object as an argument

```
Package functions;
Import java.util.function.Fucntion;
class Employee
{
     String ename;
     int  Salary;
     Employee(String ename, int Salary)
     {
          this.ename= ename;   // class elements & local variables the same
          this.Salary= Salary;
     }
Public class Demo2
{
   public static void main(String [ ] args)
    {     ArrayList <Employee> emplist =new ArrayList<Employee> ();
```

Lambda Expression

Object as an argument

```
emplist.add(new Employee("David", 50000));
emplist.add(new Employee("John", 30000));
emplist.add(new Employee("Mary", 20000));

 Function<Employee, Integer> fn= e-> {
                  int sal =e.Salary;
                  if(sal>=10000 && sal <= 20000)
                      return(sal*10/100);
                  else if(sal>20000 && sal <= 30000)
                      return(sal*20/100);
                  else if(sal>30000 && sal <= 50000)
                      return(sal*30/100);
                  else
                      return (sal*40/100);
              }
```

Lambda Expression

Object as an argument

```
for(Employee emp : emplist)
       {
          int bonus=fn.apply(emp);
          System.out.println(emp.ename + “  ” + emp.Salary);
          System.out.println(“Bonus  is:”  + bonus);
       }
// Let us add predicate Interface
    Predicate<Integer> p =b-> b>5000;
    for(Employee emp: emplist)
       {
          int bonus=fn.apply(emp);
          if(p.test(bonus)
           {
             System.out.println(emp.enmae + “  ” emp.Salary);
             System.out.println(“Bonus is :”+ bounus);
          }  }    } }
```

Output

```
David 50000
Bonus is : 15000
John 30000
Bonus is : 6000
Mary     20000
Bonus is: 2000
```

Output

```
David 50000
Bonus is : 15000
John 30000
Bonus is : 6000
```

Lambda Expression

To combine or chain functions there are two methods

Function Chaining ---- andThen( ), compose( )

Example Demo3

```
  package functions;
  import java.uitl.function.Function;
  public class Demo3
  {
     public static void main(String [ ] args)
     {
        Function <Integer, Integer>   f1 = n -> n*2;
        Function <Integer, Integer>   f2 =n->n*n*n;
        System.out.println(f1.andThen(f2).apply(2));
   /* first the first lambda expression is executed and the result will be   applied on the
     second Lambda expression and the result will be   4 – for the first LE after 64 form the
     second LE */
```

Lambda Expression

To combine or chain functions there are two methods

Function Chaing ----andThen( ), compose( )

Example Demo3

```
        System.out.println(f1.compose(f2).apply(2));
    // first the second lambda expression is executed and the result will be
     passed as an argument for the first LE
   }
}
```

Difference between Predicate and Function

Predicate- we have to pass single argument of any type and boolean type
value will be returned
- we have a method test( ) with predicate

Function – we have to pass parameter type, return type
- any parameter can be passed and any value will be returned
- we have apply ( ) method in function

Lambda Expression

Difference between Predicate and Function

Predicate- used for conditional statement and Function used to process something and get result

Predefined Functional Interface

Consumer - take a single parameter as input but it doesn’t return any value
- it has an accept( ) method

Supplier - it doesn’t take any parameter but it return any type of value or object, it has get( ) method

Example for Consumer LE

```
package consumer;
import java.util.function.Cosumer;
public class Demo1
 {
   public static void main(String [ ] args)
    {
     Cosumer<String> c= s-> System.out.println(s);
     c.accept(“Welcome”);
 }   }
```

Lambda Expression

Combine Predicate , Function and consumer LE

-work on Employee object to – calculate Bonus

– check Bonus > 5000

– print emp detail

Example Demo2

```
Package  consumer;
class Employee
{
    String ename;
    int Salary;
    String  gender;
    Employee (String ename, int Salary, String gender)
     {
        this.ename=ename;
        this.Salary=Salary;
        this.gender=gender;
     }
```

Lambda Expression

```
public class Demo2
{
   public static void main(String [ ] args)
   {
      ArrayList<Employee> emplist = new ArrayList<Employee> ( );
          emplist.add(new Employee("David", 50000, "Male"));
          emplist.add(new Employee("John", 30000, "Male"));
          emplist.add(new Employee("Mary", 20000, "Female"));
          emplist.add(new Employee("Scott", 60000, "Male"));
// Function -----task1
         Function<Employee, Integer> f= emp->(emp.salary*10)/100;
//Predicate ------ task 2
         Predicate<Integer> p = b-> b>=5000;
//Consumer ----- task 3
         Consumer<Employee> c=emp->{
                    System.out.println(emp.ename);
                    System.out.println(emp.Salary);
                    System.out.println(emp.gender);    };
```

Lambda Expression

```
    for(Employee  e:emplist)
        {
            int bonus=f.apply(e);
            if(p.test(bonus))
             {
                    c.accept(e);
                    System.out.println("Employee bonus:" +bonus);
             }
           }
         }
      }
   }
}
```

```
David
50000
Male
Employee Bonus is : 5000
Scott
Male
Employee Bonus is: 6000
```

Lambda Expression

Chaining Consumer with example

```
Package consumer;
import java.util.function.Cosumer;
public class Demo3
{
    public static void main(String [ ] args)
      {
       Consumer<String> c1= s->System.out.Println(s+ "     is white");
       Consumer<String> c2= s->System.out.Println(s+ "     has four legs");
       Consumer<String> c3= s->System.out.Println(s+ "     eating grass");
        c1.andThen(c2).andThen(c3).accept("Cow);
/* or
    Consumer<String> c4=c1.andThen(c2).andThen(c3);
       c4.accept("Cow");  */
/* or
    c1.accept("Cow"); c2.accept("Cow"); c2.accept("Cow"); */
    }
}
```

Output

```
Cow is White
Cow has four legs
Cow is eating grass
```

Lambda Expression

Supplier – is also a functional interface
- has a method get( )
- do not take input parameter but return value of any type
- we will have return type

```
Package Supplier;
import java.util.function.Supplier;
Import java.util.Date;
public class Demo1
{
   public static void main(String [ ] args)
     {
      Supplier<Date> s = ( )-> new Date( );
      System.out.println(s.get( ));
     }
}
```

Output

```
Tue Mar 28 11:45 2022
```

Object Serialization

1.4 Object Serialization

- Object serialization in Java refers to the process of converting an object's state (its data and behavior) into a byte stream, which can then be stored on disk or transmitted over a network.
- The reverse process, in which an object is reconstructed from its serialized byte stream, is called deserialization.
- To read an entire object from or write an entire object to a file, Java provides **object serialization**.

1.4 Object Serialization …

In Java, object serialization is achieved through the use of the **java.io.Serializable** interface, which is a marker interface that indicates that a class can be serialized.
To make a class serializable, you simply need to implement the **Serializable** interface, like this:

```
import java.io.Serializable;
public class Person implements Serializable {
   private String name;
   private int age;
   // other fields and methods
}
```

Declarative programming

Shall we put this topic next to functional programming: YES

Declarative programming

Declarative programming and imperative programming are two different paradigms of programming, each with its own approach to writing code.

Declarative programming is a programming paradigm

- emphasizes what the program should accomplish, rather than how it should accomplish it.
- the programmer specifies the rules, constraints, or logic that describe the desired outcome.
- That use a higher-level of abstraction, which enables the programmer to write more concise and expressive code.
- Examples of declarative programming in Java include functional programming, the Stream API, and SQL.

Declarative programming

Imperative programming, on the other hand, is a programming paradigm that emphasizes how the program should accomplish the task at hand.

Imperative programming,

- the programmer specifies a series of step-by-step instructions for the computer to execute.
- are often low-level and closer to the hardware, making them more efficient for certain types of tasks.
- Java is primarily an imperative programming language, although Java 8 introduced functional programming features that allow for a more declarative style of programming.

Declarative programming

- Overall, declarative programming is a useful approach for tasks that require a higher-level of abstraction,
- while imperative programming is a better choice for tasks that require low-level control over the computer's hardware resources.
- Both paradigms can be used effectively in Java, depending on the specific requirements of the task at hand.

Example of Declarative programming

1.Functional programming: In Java 8 and later versions, you can write functions as lambda expressions and pass them around as arguments.

- This approach emphasizes what the code should do, rather than how it should do it.
- For example, the following code uses a lambda expression to sort a list of strings in descending order:

```
List<String> names = Arrays.asList("John", "Alice", "Bob");
names.sort((s1, s2) -> s2.compareTo(s1));
```

Example of Declarative programming

2. Stream API: The Stream API in Java provides a declarative way to process collections of data.
You can chain together a series of operations, such as **filter**, **map**, and **reduce**, to specify what you want to do with the data.
For example, the following code uses the Stream API to count the number of even integers in a list:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);
long count = numbers.stream()
                    .filter(n -> n % 2 == 0)
                    .count();
```

Example of Declarative programming

3. SQL queries: You can use Java Database Connectivity (JDBC) to write declarative SQL queries to interact with databases.
The SQL queries specify what data you want to retrieve or modify, rather than how to retrieve or modify it.
For example, the following code retrieves all records from a database table:

```
String sql = "SELECT * FROM employees";
try (Connection conn = DriverManager.getConnection(url,
user,password);
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery(sql)) {
      // process the results
   }
catch (SQLException e) {
   // handle the exception
}
```