UNIT – II

Inheritance, Packages and Interfaces – Hierarchical abstractions, Base class object, subclass, subtype, substitutability, forms of inheritance specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance. Member access rules, super uses, using final with inheritance, polymorphism- method overriding, abstract classes, the Object class. Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages, differences between classes and interfaces, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces. Exploring java.io.

**SubClass, SubType, and Substitutability**

**Substitutability**

The idea of substitutability is that the type given in a declaration of a variable does not have to match the type associated with a value the variable is holding. It is very common in object-oriented programs.

When a new class are constructed using inheritance from existing classes, the argument used to justify the validity of substitutability is as follows

1. Instances of the subclass must poses all data fields associated with the parent class.
2. Instances of the subclass must implement through inheritance at least all functionality defined for the parent class.
3. Thus an instance of ac child class can mimic the behavior of the parent class and should be indistinguishable from an instance of the parent’s class if substituted in a similar situation.

**Subtype:**

The term subtype is used to describe the relationship between types that explicitly recognizes the principle of substitution. That is a type B is considered to be a subtype of A if two conditions holds.

1. An instance of B can legally be assigned to a variable declared as type A. and
2. This value can then be used by the variable with no observable change in behavior

**Subclass:**

the term subclass refers merely to the mechanics of constructing a new class using inheritance, and is easy to recognize from source description of a program by the presence of the keyword ***extends***.

**Forms of inheritance**

1. Specialization
2. Specification
3. Construction
4. Extension
5. Limitation
6. Combination

**Specialization**

The child class is a special case of the parent, in other words, the child class is a subtype of the parent class.

The creation of application window classes using inheritance from the java library class Frame is an example of sub classification for specialization.

**Public class PinBallGame extends Frame**

**{**

**…….**

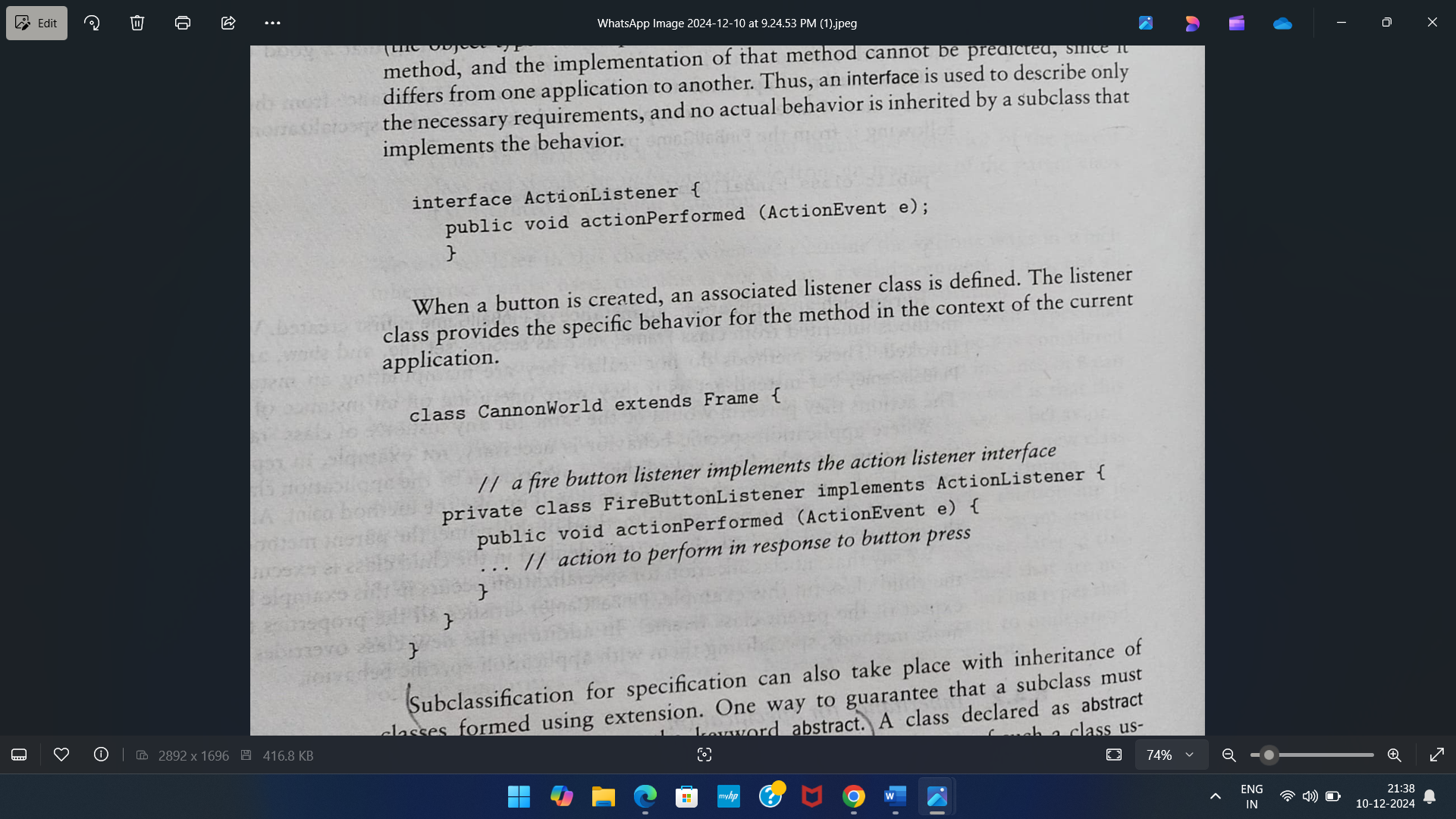
**}**

To run such an application, an instance of PinBallGame is first created. Various methods inherited from class Frame, such as setSize, setTitle and show are then invoked. These methods do not realize they are manipulating an instance of PinBallGame, but instead act as if they were operating on an instance of Frame. The actions they perform would be the same for any instance of Frame.

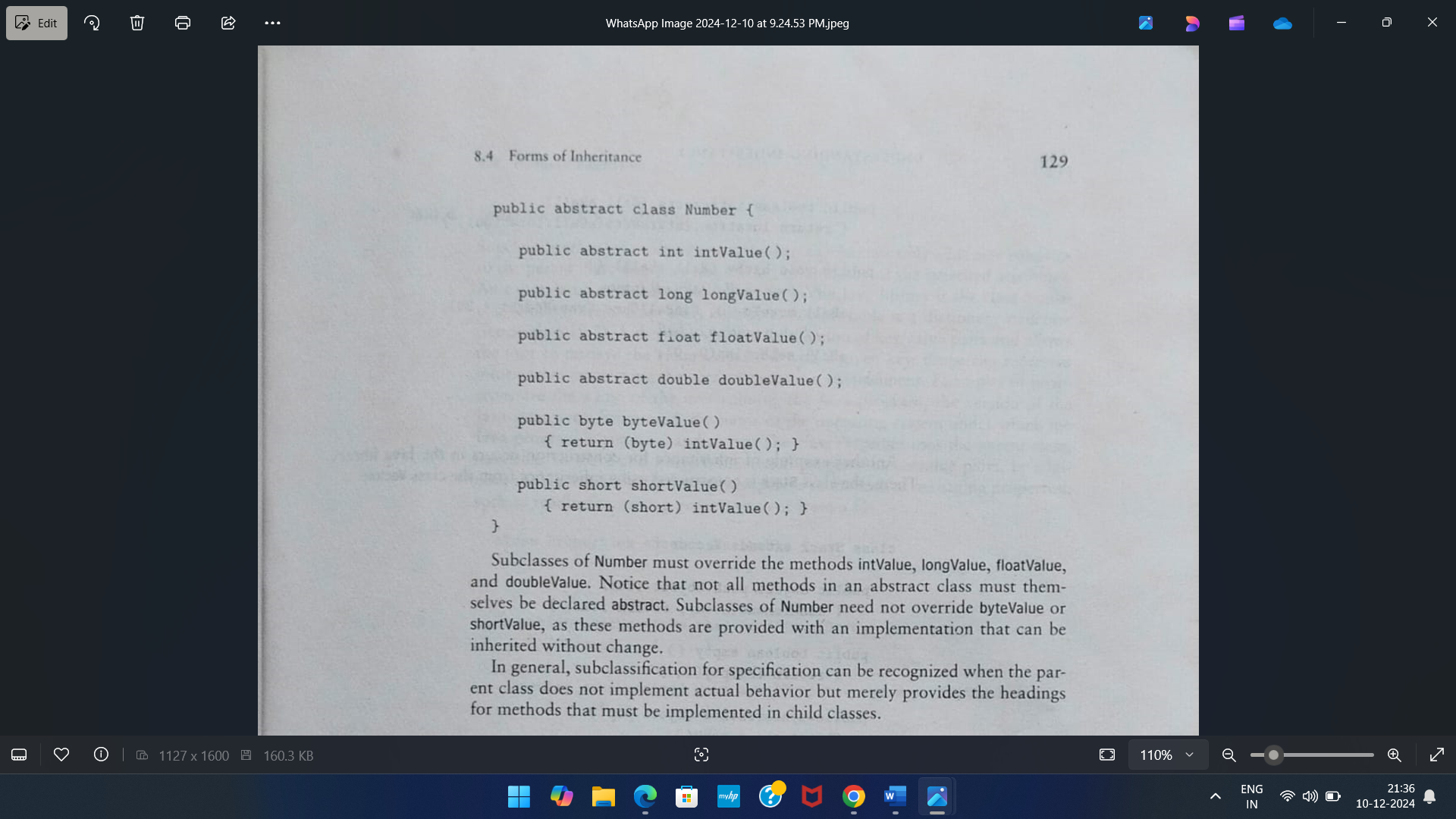
**Specification**

The parent class defines behavior that is implemented in the child class but not in the parent class.

There are two different mechanisms provided by the java language to support the idea of inheritance of specification. The most obvious technique is the use of interfaces. We have seen examples of this in the way that events are handled by the java library. For instance , the characteristics needed for an ActionListener can be described by a single method, and the implementation of that method cannot be predicted, since it differs from one application to another. Thus, an interface is used to describe only the necessary requirements, and no actual behavior is inherited by a subclass that implements the behavior.



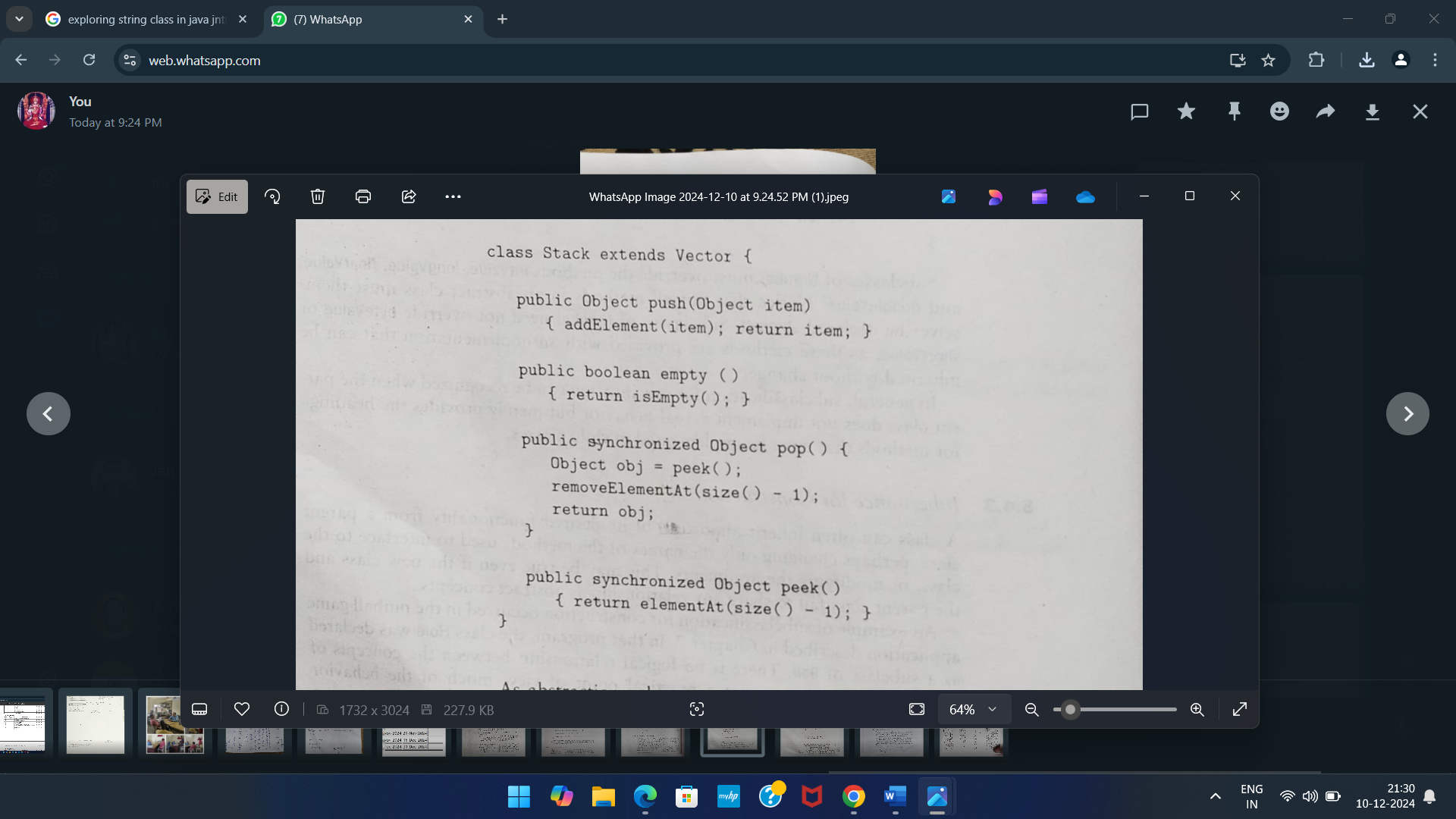
Another example, abstract class in the java library is Number, apparent class for the numeric wrapper classes Integer, Long, Double and so on. The class description is as follows



**Construction**

The child class makes use of the behavior provided by the parent class but is not a subtype of the parent class.

One example of inhertance for construction occurs in the java liobrary, there the class Stack is constructed using inheritance from the class Vector.

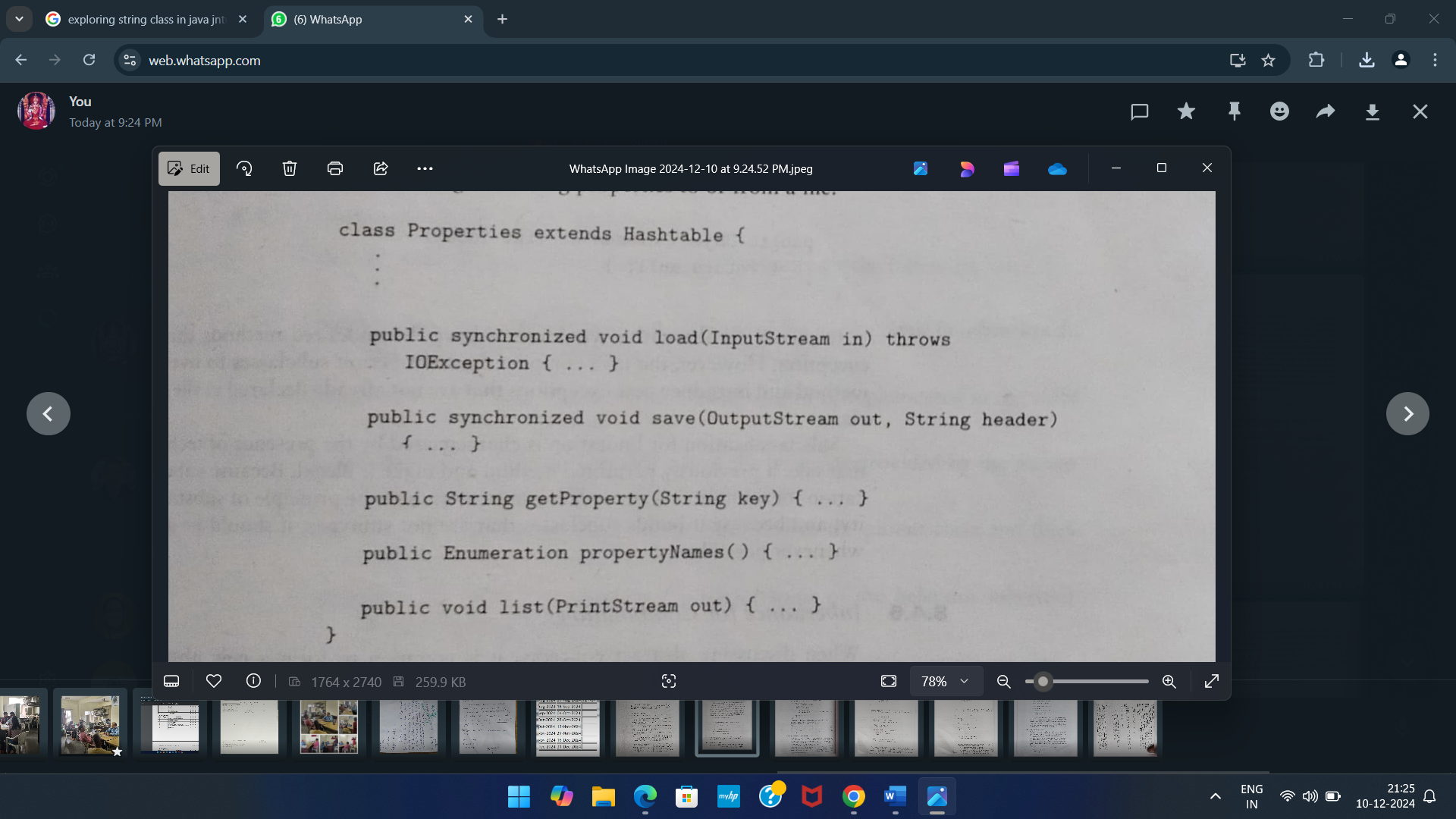


As abstraction, the concept of the stack and the concept of a vector have little in common; however, from a pragmatic point of view using the vector class as apparent greatly simplifies the implementation of the stack.

**Extension**

The child class adds new functionality to the parfent class, but does not change any inherited behavior.

An example of inheritance for extension in the java library is the class Properties, which inherits from the class Hashtable. A hashtable is dictionary structure. A dictionary stores a collection of key/value pairs and allows the user to retrieve the value asscioated with a given key. Properties represent information concerning the current execution environment. Example of properties are name of he user running the java program, the version of the java interpreter being used, the name of the operating system under which the java program is running and so on. The class properties uses the parent class, Hashtable, to store and retrieve the actual property name/value pairs.



**Limitaion**

The child class restricts the use of some of the behavior inherited from type parent class.

Although there are no examples of subclassification for limitation in the java library. We could imagine the following. Suppose you wanted to create the class Set, in a fashion similar to the way the class Stack is subclassed from Vector. Say you also wanted to ensure that only Set operations were used on the set and not the vector operations. One way to accomplish this would be to override the undesired methods so that if they were executed they would produce obviously incorrect results, or print message indicating they should not be used.

Class Set extends Vector

{

Public int indexOf(Object obj)

{

System.out.println(“Do not use Set. indexOf”);

}

Public Object elementAt(int index)

{

return null;

}}

**Combination**

The child class inherits features from more than one parent class. Although multiple inheritance is not supported directly by java, it can be simulated in part by classes that use both inheritance and implementation of an interface or implements two or more interfaces.

Although the java language does not permit a subclass to be formed by inheritance from more than one parent class, several approximation to the concept are possible . for example, it is common for a new class to both extends an existing class and implement an interface.

Public Hole extends Ball implements PinBallTarget

{

}

It is also possible for classes to implements more than one interface, and thus be viewed as a combination of two categories.

**Super keyword**

If we create an object to super class, we can access only the super class members, but not the sub class members. but if we create the subclass object, all the members of both super and sub classes are available to it. This is the reason, we always create an object to sub class in inheritance. Some times, the super class members and sub class members may have same names. In that case by default only sub classes ,members are accessible. This shown in the following example program.

**Example Program -1**

class One

{

int i=10;

void show()

{

System.out.println("super class method i="+i);

}

}

class Two extends One

{

int i=20;

void show()

{

System.out.println("sub class meyhod i="+i);

}

}

class Super1

{

public static void main(String args[])

{

Two t=new Two();

t.show();

}

}

Whenever a sub class needs to refer to its immediate super class, irt can do so by use of the keyword super.

**Super keyword can be used to refer to**

1. Super class constructor.
2. Super class members(data members or methods)

**Referring super class constructor**

A subclass can call a constructor defined by its super class by use of the following from of super

***super(arg-list).***

Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass’ constructor.

**Example Program -2**

class One

{

int i;

One(int i)

{

this.i=i;

}

}

class Two extends One

{

int i;

Two(int a,int b)

{

super(a);

i=b;

}

void show()

{

System.out.println("sub class i="+i);

System.out.println("super class i="+super.i);

}

}

class Super2

{

public static void main(String args[])

{

Two t=new Two(10,20);

t.show();

}

}

**Referring super class members**

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.

This usage has the following general form:

***super.member***

Here, member can be either a method or an instance variable

**Example Program -3**

class One

{

int i;

One(int i)

{

this.i=i;

}

}

class Two extends One

{

int i;

Two(int a,int b)

{

super(a);

i=b;

}

void show()

{

System.out.println("sub class i="+i);

System.out.println("super class i="+super.i);

}

}

class Super2

{

public static void main(String args[])

{

Two t=new Two(10,20);

t.show();

}

}

**Final with inheritance**

#### Java final keyword

In java, the final is a keyword and it is used with the following things.

* **With variable (to create constant)**
* **With method (to avoid method overriding)**
* **With class (to avoid inheritance)**

1. final with variables

When a variable defined with the final keyword, it becomes a constant, and it does not allow us to modify the value. The variable defined with the final keyword allows only a one-time assignment, once a value assigned to it, never allows us to change it again.

final int a = 10;

1. **Using final to Prevent Overriding**

There will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

The following fragment illustrates final:

**class A**

**{**

**final void meth()**

**{**

**System.out.println("This is a final method.");**

**}**

**}**

**class B extends A**

**{**

**void meth()**

**{ // ERROR! Can't override. System.out.println("Illegal!");**

**}**

**}**

Because meth( ) is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result

1. **Using final to Prevent Inheritance**

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too.

Here is an example of a final class:

final class A

{

// ...

}

**// The following class is illegal.**

class B extends A

{

// ERROR! Can't subclass A // ...

}

#### Java Polymorphism

* The polymorphism is the process of defining same method with different implementation.
* That means creating multiple methods with different behaviors.
* In java, polymorphism implemented using method overloading and method overriding.

Ad hoc polymorphism

The ad hoc polymorphism is a technique used to define the same method with different implementations and different arguments. In a java programming language, ad hoc polymorphism carried out with a method overloading concept. In ad hoc polymorphism the method binding happens at the time of compilation. Ad hoc polymorphism is also known as compile-time polymorphism. Every function call binded with the respective overloaded method based on the arguments. The ad hoc polymorphism implemented within the class only.

Pure polymorphism

The pure polymorphism is a technique used to define the same method with the same arguments but different implementations. In a java programming language, pure polymorphism carried out with a method overriding concept. In pure polymorphism, the method binding happens at run time. Pure polymorphism is also known as run-time polymorphism. Every function call binding with the respective overridden method based on the object reference. When a child class has a definition for a member function of the parent class, the parent class function is said to be overridden.

The pure polymorphism implemented in the inheritance concept only

**Dynamic method Dispatch**

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at runtime. Rather than compile time. Dynamic method dispatch is important because this how java implements run-time polymorphism. **One of the important principle in java is *a super class reference variable can refer to a sub class object*.** java uses this fact to resolve calls to overridden methods at runtime. When an overridden method is called through a super class reference. Java determines which version of that method to execute based upon the type of the object being referenced to at the call occurs. Thus this determination is made at runtime.

// Dynamic Method Dispatch

class A

{

void callme()

{

System.out.println("Inside A's callme method");

} }

class B extends A

{

// override callme()

void callme() { System.out.println("Inside B's callme method");

} }

class C extends A

{

// override callme()

void callme()

{ System.out.println("Inside C's callme method");

}

}

class Dispatch

{

public static void main(String args[])

{

A a = new A(); // object of type A

B b = new B(); // object of type B

C c = new C(); // object of type C

A r; // obtain a reference of type

A r = a; // r refers to an A object

r.callme(); // calls A's version of callme

r = b; // r refers to a B object

r.callme(); // calls B's version of callme

r = c; // r refers to a C object

r.callme(); // calls C's version of callme

} }

**/\* programs on run time polymorphism using method overriding\*/**

class Figure

{

double dim1;

double dim2;

Figure(double a,double b)

{

dim1=a;

dim2=b;

}

double area()

{

System.out.println("Area of the figure undefined");

return 0;

}

}

class Rectangle extends Figure

{

Rectangle(double a,double b)

{

super(a,b);

}

double area()

{

System.out.println("Inside Area of the rectangle");

return dim1 \*dim2;

}}

class Triangle extends Figure

{

Triangle(double a,double b)

{

super(a,b);

}

double area()

{

System.out.println("Inside Area of the triangle");

return dim1 \*dim2/2;

}}

class FindAreas

{

public static void main(String args[])

{

Figure f=new Figure(10,20);

Rectangle r=new Rectangle(20,20);

Triangle t=new Triangle(30,20);

Figure figref;

figref=r;

System.out.println("Area is"+figref.area());

figref=f;

System.out.println("Area is"+figref.area());

figref=t;

System.out.println("Area is"+figref.area());

}

}

**Final keyword with Inheritance**

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance.

**Abstract class**

An abstract class is a class that contains 0 or more abstract methods.

(or)

A class that is declared using “abstract” keyword is known as abstract class. It may or may not include abstract methods which means in abstract class you can have concrete methods (methods with body) as well along with abstract methods( without an implementation, without braces, and followed by a semicolon). An abstract class can not be **instantiated** (you are not allowed to create **object** of Abstract class) but we can create reference for the abstract class.

**Abstract class declaration**

Specifying **abstract keyword** before the class during declaration, makes it abstract.

**Abstract Method** An abstract method is a method without method body. An abstract method is written when same method has to perform different tasks depending on the object calling it.

**Syntax of abstract method:**

abstract *type name(parameter-list)*;

**Points to remember about abstract method:**  
1) Abstract method has no body.  
2) Always end the declaration with a **semicolon**(;).  
3) It must be [overridden](https://beginnersbook.com/2014/01/method-overriding-in-java-with-example/). An abstract class must be extended and in a same way abstract method must be overridden.  
4) Abstract method must be in a abstract class.

**Note:** The class which is extending abstract class must override (or implement) all the abstract methods. otherwise the class itself becomes abstract.

**Example program**

abstract class Car

{

int regno;

Car(int r)

{

regno=r;

}

void openTank()

{

System.out.println("Fill the tank");

}

abstract void steering(int direction, int angle);

abstract void braking(int force);

}

class Maruti extends Car

{

Maruti(int regno)

{

super(regno);

}

void steering(int direction,int angle)

{

System.out.println(" Take turn");

System.out.println("This is an ordinary steering");

}

void braking(int force)

{

System.out.println("Brakes applied");

System.out.println("These are hydraulic brakes" );

}

}

class Santro extends Car

{

Santro(int regno)

{

super(regno);

}

void steering(int direction,int angle)

{

System.out.println(" Take turn");

System.out.println("This car uses power steering ");

}

void braking(int force)

{

System.out.println("Brakes applied");

System.out.println("This car uses gas brakes" );

}

}

class Usecar

{

public static void main(String args[])

{

Maruti m=new Maruti(1901);

Santro s=new Santro(5001);

Car ref;

ref=m;

ref.openTank();

ref.braking(500);

ref.steering(1,300);

}

}

#### Java Object Class

In java, the Object class is the super most class of any class hierarchy. The Object class in the java programming language is present inside the **java.lang** package.Every class in the java programming language is a subclass of Object class by default.The Object class is useful when you want to refer to any object whose type you don't know. Because it is the superclass of all other classes in java, it can refer to any type of object.

Methods of Object class

The following table depicts all built-in methods of Object class in java.

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Return Value** |
| getClass() | Returns Class class object | object |
| hashCode() | returns the hashcode number for object being used. | int |
| equals(Object obj) | compares the argument object to calling object. | boolean |
| clone() | Compares two strings, ignoring case | int |
| concat(String) | Creates copy of invoking object | object |
| toString() | eturns the string representation of invoking object. | String |
| notify() | wakes up a thread, waiting on invoking object's monitor. | void |
| notifyAll() | wakes up all the threads, waiting on invoking object's monitor. | void |
| wait() | causes the current thread to wait, until another thread notifies. | void |

**Packages**

**Definition**:-Packages are containers for classes that are used to keep the class name space compartmentalized.

**(or)**

A package represents a directory that contains related group of classes and interfaces. We write the statement like

**import java.io.\*;**

We are importing classes of java.io package. Here java is a directory name and io is another sub directory within it. And the \* represents all the classes and interfaces of that io sub directory.

java Directory

java.io io sub directory

java.lang

awt directory

java.awt

event directory

java.awt.event

**Advantages of packages**

1. Packages are useful to arrange relates classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example in java all the classes and interfaces which perform input and output operation are stored in java.io package.
2. The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different classes. For example there is a Date class in **java.util** package and also there is another Date class available in **java.sql**  package.
3. A group of packages is called library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy.

**Different types of packages**

There are two different types of packages

1. Built-in packages
2. User-defined packages.

**Built–in packages**

These are the packages which are already available in java language. These packages provide all most all necessary classes, interfaces and methods for the programmer to perform any task in his program, foe everything there is a method available in java and that method can be used by the programmer without developing the logic on his own. This makes programming easy. Some of the important packages in java SE are

1. **java.lang**:- lang stands for language. This package got primary classes and interfaces essential for developing a basic java program. It consists of wrapper classes which are useful to convert primitive type to objects.
2. **java.io**:- io stands for input and output. This package contains streams. Streams are useful to store data in the for of files and also to perform input and output related tasks.
3. **java.util** :- util stands for utility. This package contains useful classes and interface like Stack. LinkedList, Vector, Arrays etc. these classes are called as collections. There are also classes for handling date and time operations.

**User-defined packages:**

Just like Built-in packages the use of java language can also create their own packages. They are called User-defined packages. User-defined packages can also be imported into other classes and used exactly in the same way as in Built–in packages.

**Defining a package**

To create a package is quite easy: simply include a package command as the first statement. In a java source file. Ant classes declared within that file will belong to the specified packages. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name.

General form of the package statement

**package pkg;**

Here , pkg is the name of the package. For example the following statement creates a package called **MyPackage.**

**Package mypackage;**

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

**Remember that case is significant, and the directory name must match the package name** exactly.

we can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

**package *pkg1*[.*pkg2*[.*pkg3*]];**

A package hierarchy must be reflected in the file system of your Java development

system. For example, a package declared as

**package java.awt.image;**

needs to be stored in **java\awt\image** in a Windows environment.

**Access Specifiers**

An access modifier is a keyword that specifies how to access the members of a class or a class itself. We can use access modifier before a class and its members. There are four access modifier available in java.

* 1. **Private :-** private members of a class are not accessible any where outside the class. They are accessible only within the class by the methods of that class.
  2. **Public:**  public members of class are accessible every where outside the class. So any other program can read them and use them.
  3. **Protected:** protected members of a class are accessible only to classes in same package but outside the class, but only to classes that subclass your class directly.
  4. **Default:** if no access modifier is written by the programmer then the java compiler uses a default access modifier. Default members are accessible outside the class, but within the same package(directory).

Another Package

Same Package

class

**Class C**

****

****

**In subclass**

**Class B**

****

****

****

**Class A**

Private

Public

Protected

default

**+**

Private members of class A are not available to class B or class C. this means any private member’s scope is limited only to that class where it is defined. so scope of private access specifier is class scope.

1. Public members of class a are available to class B and also class C. this means public members are available every where and their scope is global scope.
2. Protected members of class A are available to class B, but not in class c. but if class C is a sub class of class A then the protected members of class A are available to class c. so protected access specifier acts as public with respect to sub class.
3. When no access specifier is used, it is taken as default specifier. Default members of class A are accessible to class B which is within the same package. They are not available to class C. this means the scope of default members is package scope.

**Finding Packages and CLASSPATH:**

Java run-time system checks packages that we create in 3 ways. They are

1. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
2. Second, you can specify a directory path or paths by setting the CLASSPATH environmental variable.
3. We can use the **–classpath** option with java and javac to specify the path to your classes.

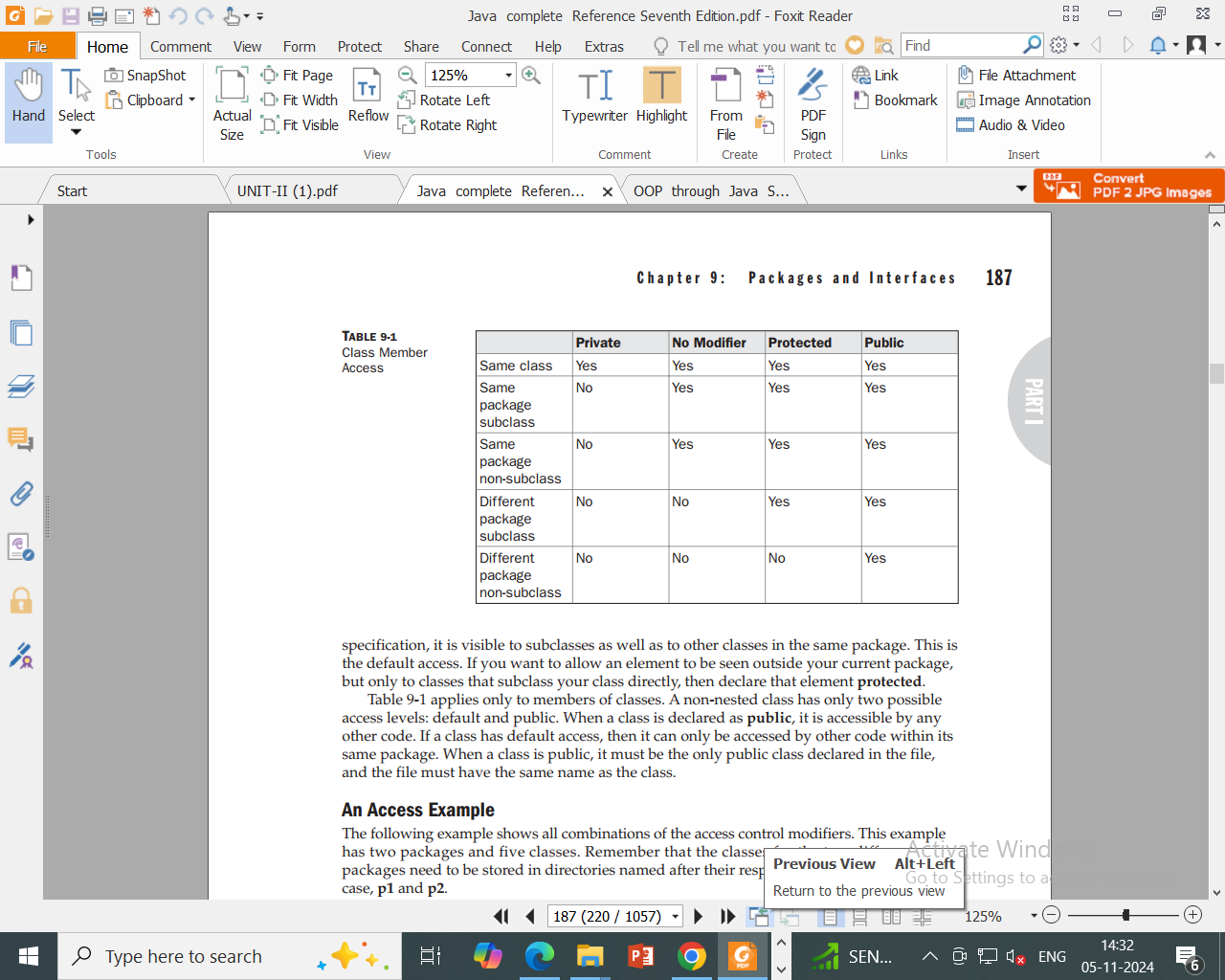
**Access Protection**

Packages are used not only to avoid class name collision but also access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code.

Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

1. Subclasses in the same package
2. Non-subclasses in the same package
3. Subclasses in different packages
4. Classes that are neither in the same package nor subclasses.



package p1;

public class Protection {

int n = 1;

private int n\_pri = 2;

protected int n\_pro = 3;

public int n\_pub = 4;

public Protection() {

System.out.println("base constructor");

System.out.println("n = " + n);

System.out.println("n\_pri = " + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pub = " + n\_pub);

}}

package p1;

class Derived extends Protection {

Derived() {

System.out.println("derived constructor");

System.out.println("n = " + n);

// class only

// System.out.println("n\_pri = "4 + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pub = " + n\_pub);

}

}

package p1;

class SamePackage {

SamePackage() {

Protection p = new Protection();

System.out.println("same package constructor");

System.out.println("n = " + p.n);

// class only

// System.out.println("n\_pri = " + p.n\_pri);

System.out.println("n\_pro = " + p.n\_pro);

System.out.println("n\_pub = " + p.n\_pub);

}

}

package p1;

// Instantiate the various classes in p1.

public class Demo {

public static void main(String args[]) {

Protection ob1 = new Protection();

Derived ob2 = new Derived();

SamePackage ob3 = new SamePackage();

}

}

package p2;

class Protection2 extends p1.Protection {

Protection2() {

System.out.println("derived other package constructor");

// class or package only

// System.out.println("n = " + n);

// class only

// System.out.println("n\_pri = " + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pub = " + n\_pub);

}

}

This is file OtherPackage.java:

package p2;

class OtherPackage {

OtherPackage() {

p1.Protection p = new p1.Protection();

System.out.println("other package constructor");

// class or package only

// System.out.println("n = " + p.n);

// class only

// System.out.println("n\_pri = " + p.n\_pri);

// class, subclass or package only

// System.out.println("n\_pro = " + p.n\_pro);

System.out.println("n\_pub = " + p.n\_pub);

}

}

**// Demo package p2.**

package p2;

// Instantiate the various classes in p2.

public class Demo {

public static void main(String args[]) {

Protection2 ob1 = new Protection2();

OtherPackage ob2 = new OtherPackage();

}

}

**Importing Packages in java**

in java, the ***import*** keyword used to import built-in and user-defined packages. When a package has imported, we can refer to all the classes of that package using their name directly.

The import statement must be after the package statement, and before any other statement.

Using an import statement, we may import a specific class or all the classes from a package.

## importing specific class

Using an importing statement, we can import a specific class. The following syntax is employed to import a specific class.

**Syntax**

Import packageName.ClassName;

## Importing all the classes

Using an importing statement, we can import all the classes of a package. To import all the classes of the package, we use \* symbol. The following syntax is employed to import all the classes of a package.

**Syntax**

import packageName.\*;

package MyPack;

public class Balance {

String name;

double bal;

public Balance(String n, double b) {

name = n;

bal = b;

}

public void show() {

if(bal<0)

System.out.print("--> ");

System.out.println(name + ": $" + bal);

}

}

import MyPack.\*;

class TestBalance {

public static void main(String args[]) {

/\* Because Balance is public, you may use Balance

class and call its constructor. \*/

Balance test = new Balance("J. J. Jaspers", 99.88);

test.show(); // you may also call show()

}

}

**Interface in java**

Using the keyword interface, you can fully abstract a class interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don‘t make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. Each class can determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the ―one interface, multiple methods‖ aspect of polymorphism.

**Interfaces are designed to support dynamic method resolution at run time.**

Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and no extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.

**Defining an Interface:**

* An interface is defined much like a class.
* This is the general form of an

interface: access interface name

{

return-type method-name1(parameter-list);

return-type method-name2(parameter-list);

type final-varname1 = value;

type final-varname2 = value;

return-type method-nameN(parameter-list);

type final-varnameN = value;

}

Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default

implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

An example of an interface definition. It declares a simple interface which contains one method called **callback( )** that takes a single integer parameter.

**interface Callback**

**{**

**void callback(int param);**

**}**

**Implementing Interfaces:**

Once an interface has been defined, one or more classes can implement that interface.

To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

**The general form of a class that implements the interface:**

**access class classname [extends superclass][implements interface [,interface...]] {**

* **class-body**

**}**

Here, access is either public or not used. If a class implements more than one interface,the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

**Example**: class that implements the Callback interface shown earlier.

class Client implements Callback

{

public void callback (int p)

{

System.out.println("callback called with " + p);

}}

Notice that callback( ) is declared using the public access specifier.When you implement an interface method, it must be declared as public.

For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

class Client implements Callback

{

public void callback(int p)

{

System.out.println("callback called with " + p);

}

void nonIfaceMeth()

{

System.out.println("Classes that implement interfaces"+"may also define other members, too.");}}

**Accessing Implementations Through Interface References:**

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the ―callee.‖ This process is similar to using a superclass reference to access a subclass object.

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the callback( ) method via an interface reference variable:

class TestIface

{

public static void main(String args[])

{

Callback c = new Client();

c.callback(42);

}

}

**Output:**

callback called with 42.

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its interface declaration.

Thus, c could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

The preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

// Another implementation of Callback.

class AnotherClient implements Callback

{

public void callback(int p)

{

System.out.println("Another version of callback"); System.out.println("p squared is " + (p\*p)); }

}

class TestIface2

{

public static void main(String args[])

{

Callback c = new Client();

AnotherClient ob = new AnotherClient();

c.callback(42);

c = ob; // c now refers to AnotherClient object

c.callback(42);

}

}

**Output:**

callback called with 42

Another version of callback

p squared is 1764

As you can see, the version of callback( ) that is called is determined by the type of object that c refers to at run time.

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

**For example:**

abstract class Incomplete implements Callback

{

int a, b;

void show()

{

System.out.println(a + " " + b);

}

}

Here, the class Incomplete does not implement callback( ) and must be declared as abstract. Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

**Applying Interfaces:**

We define a stack interface, leaving it to each implementation to define the specifics. Let‘s look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called IntStack.java. This interface will be used by both stack implementations. Example:

* Define an integer stack interface. interface IntStack

{

void push(int item); // store an item int pop(); // retrieve an item

}

The following program creates a class called FixedStack that implements a fixed-length version of an integer stack:

Example:

* An implementation of IntStack that uses fixed storage.

class FixedStack implements IntStack

{

private int stck[];

private int tos;

FixedStack(int size)

stck = new int[size];

tos = -1;

}

* Push an item onto the stack public void push(int item)

{

if(tos==stck.length-1) // use length member

System.out.println("Stack is full.");

else

stck[++tos] = item;

}

* Pop an item from the stack

public int pop()

{

if(tos < 0)

{

System.out.println("Stack underflow."); return 0;

}

else

return stck[tos--];

}

}

class IFTest

{

public static void main(String args[])

{

FixedStack mystack1 = new FixedStack(5);

* push some numbers onto the stack

for(int i=0; i<5; i++)

mystack1.push(i);

for(int i=0; i<8; i++)

mystack2.push(i);

* pop those numbers off the stack

System.out.println("Stack in mystack1:");

for(int i=0; i<5; i++)

System.out.println(mystack1.pop());

System.out.println("Stack in mystack2:");

for(int i=0; i<8; i++)

System.out.println(mystack2.pop());

}

}

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled. Example:

* Implement a "growable" stack. class DynStack implements IntStack

{

private int stck[];

private int tos;

* allocate and initialize stack DynStack(int size)

{

stck = new int[size];

tos = -1;

}

* Push an item onto the stack public void push(int item)

{

* if stack is full, allocate a larger stack if(tos==stck.length-1)

{

int temp[] = new int[stck.length \* 2];

// double size

for(int i=0; i<stck.length; i++)

temp[i] = stck[i];

stck = temp;

stck[++tos] = item;

}

else

stck[++tos] = item;

}

* Pop an item from the stack

public int pop()

{

if(tos < 0) {

System.out.println("Stack underflow.");

return 0;

}

else

return stck[tos--];

}

}

class IFTest2

{

public static void main(String args[])

{

DynStack mystack1 = new DynStack(5);

DynStack mystack2 = new DynStack(8);

* these loops cause each stack to grow

for(int i=0; i<12; i++)

mystack1.push(i);

for(int i=0; i<20; i++)

mystack2.push(i);

System.out.println("Stack in mystack1:");

for(int i=0; i<12; i++)

System.out.println(mystack1.pop());

System.out.println("Stack in mystack2:");

for(int i=0; i<20; i++)

System.out.println(mystack2.pop());

}

}

The following class uses both the FixedStack and DynStack implementations.It does so through an interface reference. This means that calls to push( ) and pop( )are resolved at run time rather than at compile time.

/\* Create an interface variable and access stacks through it.

\*/

class IFTest3

{

public static void main(String args[])

{

IntStack mystack; // create an interface reference variable DynStack ds = new DynStack(5);

FixedStack fs = new FixedStack(8);

mystack = ds; // load dynamic stack

* push some numbers onto the stack for(int i=0; i<12; i++) mystack.push(i);

mystack = fs;

// load fixed stack

for(int i=0; i<8; i++)

mystack.push(i);

mystack = ds;

System.out.println("Values in dynamic stack:"); for(int i=0; i<12; i++) System.out.println(mystack.pop());

mystack = fs;

System.out.println("Values in fixed stack:"); for(int i=0; i<8; i++) System.out.println(mystack.pop());

}

}

In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push( ) and pop( ) defined by the DynStack implementation. When it refers to fs, it uses the versions of push( ) and pop( ) defined by FixedStack.

These determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

**Variables in Interfaces:**

Interfaces can be used to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When we include that interface in a class (that is, when you ―implement‖ the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations. If an interface contains no methods, then any class that includes such an interface doesn‘t actually implement anything. It is as if that class were importing the constant variables into the class name space as final variables

**Interfaces Can Be Extended:**

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interfaceinheritance chain.

Example:

* One interface can extend another. interface A

{

void meth1(); void meth2();

}

* B now includes meth1() and meth2() -- it adds meth3(). interface B extends A

{

void meth3();

}

* this class must implement all of A and B

class MyClass implements B

{

public void meth1()

{

System.out.println("Implement meth1().");

}

public void meth2()

{

System.out.println("Implement meth2().");

}

public void meth3()

{

System.out.println("Implement meth3().");

}

}

class IFExtend

{

public static void main(String arg[])

{

MyClass ob = new MyClass();

ob.meth1();

ob.meth2();

ob.meth3();

}}

Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

**Super keyword**

If we create an object to super class, we can access only the super class members, but not the sub class members. but if we create the subclass object, all the members of both super and sub classes are available to it. This is the reason, we always create an object to sub class in inheritance. Some times, the super class members and sub class members may have same names. In that case by default only sub classes ,members are accessible. This shown in the following example program.

**Example Program -1**

class One

{

int i=10;

void show()

{

System.out.println("super class method i="+i);

}

}

class Two extends One

{

int i=20;

void show()

{

System.out.println("sub class meyhod i="+i);

}

}

class Super1

{

public static void main(String args[])

{

Two t=new Two();

t.show();

}

}

Whenever a sub class needs to refer to its immediate super class, irt can do so by use of the keyword super.

**Super keyword can be used to refer to**

1. Super class constructor.
2. Super class members(data members or methods)

**Referring super class constructor**

A subclass can call a constructor defined by its super class by use of the following from of super

***super(arg-list).***

Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass’ constructor.

**Example Program -2**

class One

{

int i;

One(int i)

{

this.i=i;

}

}

class Two extends One

{

int i;

Two(int a,int b)

{

super(a);

i=b;

}

void show()

{

System.out.println("sub class i="+i);

System.out.println("super class i="+super.i);

}

}

class Super2

{

public static void main(String args[])

{

Two t=new Two(10,20);

t.show();

}

}

**Referring super class members**

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used.

This usage has the following general form:

***super.member***

Here, member can be either a method or an instance variable

**Example Program -3**

class One

{

int i;

One(int i)

{

this.i=i;

}

}

class Two extends One

{

int i;

Two(int a,int b)

{

super(a);

i=b;

}

void show()

{

System.out.println("sub class i="+i);

System.out.println("super class i="+super.i);

}

}

class Super2

{

public static void main(String args[])

{

Two t=new Two(10,20);

t.show();

}

}