Inheritance, Polymorphism, Interfaces

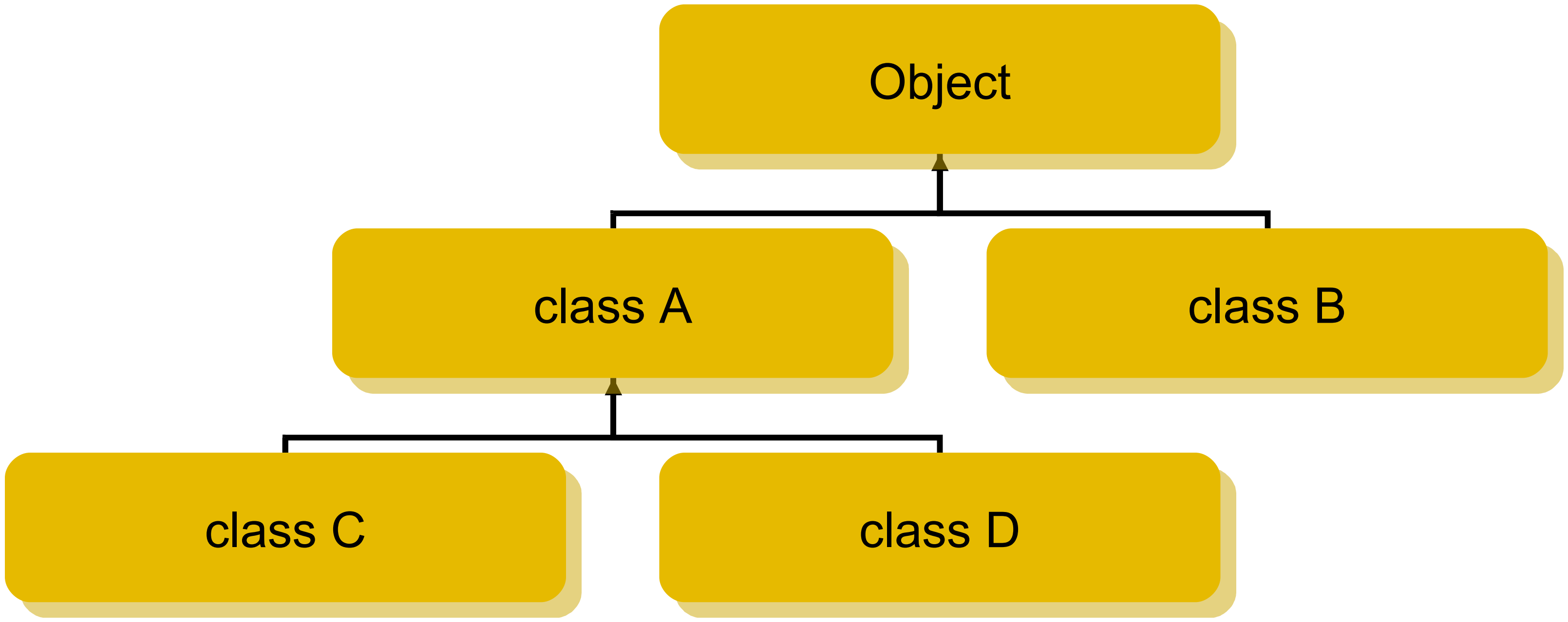
Objectives

- At the end of the lesson, the student should be able to:
  - Define super classes and subclasses
  - Override methods of super classes
  - Create final methods and final classes

Inheritance

Inheritance

- In Java, all classes, including the classes that make up the Java API, are subclassed (extended) from the Object super class.
- A sample class hierarchy is shown below.



Super class & Sub class

- Super class (Parent class)
  - Any class above a specific class in the class hierarchy.
- Sub class (Child class)
  - Any class below a specific class in the class hierarchy.

Reusability

- Benefits of Inheritance in OOP : Reusability
  - Once a behavior (method) is defined in a super class, that behavior is automatically inherited by all subclasses
  - Thus, you can encode a method only once and they can be used by all subclasses.
  - Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all subclasses
  - A subclass only needs to implement the differences between itself and the parent.

Inheritance:
How to derive a sub class

extends keyword

- To derive a child class, we use the extends keyword.
- Suppose we have a parent class called Person.

```
public class Person {
        protected String name;
        protected String address;
        /**
        * Default constructor
        */
        public Person(){
        System.out.println(“Inside Person:Constructor”);
        name = ""; address = "";
        }
        . . . .
        }
```

extends keyword

- Now, we want to create another class named Student
- Since a student is also a person, we decide to just extend the class Person, so that we can inherit all the properties and methods of the existing class Person.
- To do this, we write,

```
public class Student extends Person {

  public Student(){
  System.out.println( “Inside Student:Constructor”);
  }
  . . . .
}
```

Inheritance:
Constructor Calling Chain

How Constructor method of a Super class gets called

- When a Student object, a subclass (child class), is instantiated, the default constructor of its super class (parent class), Person class, is invoked implicitly to do the necessary initializations.
- After that, subclass's constructor method is then invoked

Example: Constructor Calling Chain

- To illustrate this, consider the following code,
- In the code, we create an object of class Student.

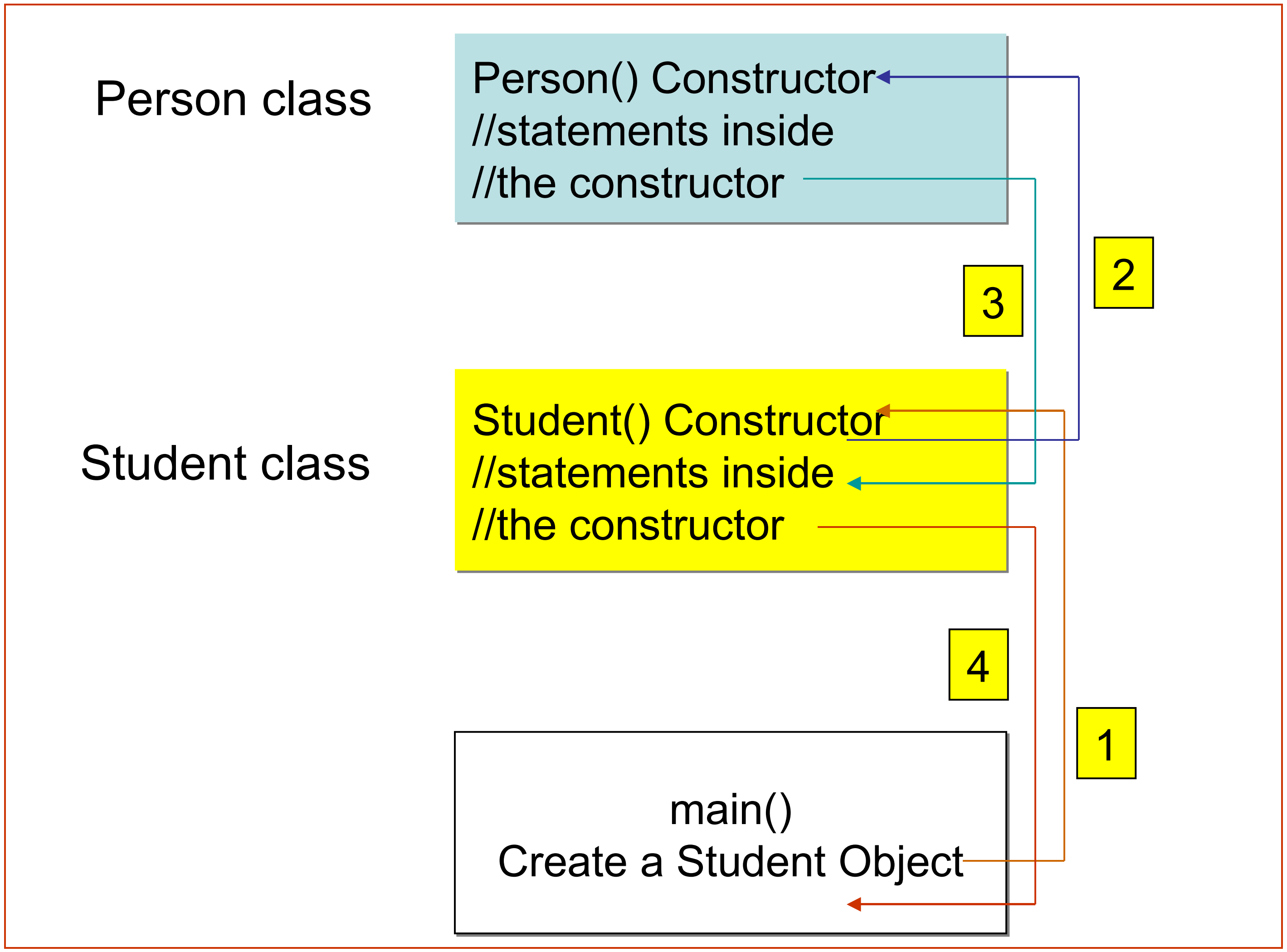
```
public static void main( String[] args ){
Student anna = new Student();
}
```

- The output of the program is,

```
Inside Person:Constructor
Inside Student:Constructor
```

Constructor Calling Chain

The program flow is shown below



Inheritance: “super” Keyword

The “super” keyword

- A subclass can also explicitly call a constructor of its immediate super class.

- This is done by using the *super* constructor call.

- A super constructor call in the constructor of a subclass will result in the execution of relevant constructor from the super class, based on the arguments passed.

The "super" keyword

- For example, given our previous example classes Person and Student, we show an example of a super constructor call.
- Given the following code for Student,

```
public Student(){
super( "SomeName", "SomeAddress" );
System.out.println("Inside Student:Constructor");
}
```

The “super” keyword

- Few things to remember when using the super constructor call:
  - The super() call MUST OCCUR AS THE FIRST STATEMENT IN A CONSTRUCTOR.
  - The super() call can only be used in a constructor definition.

The “super” keyword

- Another use of super is to refer to members of the super class (just like the *this* reference).
- For example,

```
public Student() {
super.name = “somename”;
super.address = “some address”;
}
```

Inheritance:
Overriding methods

Method Overriding

- Definition
  - In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
  - When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass.
  - The version of the method defined by the superclass will be hidden.

Overriding methods

- If for some reason a derived class needs to have a different implementation of a certain method from that of the super class, overriding methods could prove to be very useful.
- A subclass can override a method defined in its super class by providing a new implementation for that method.

Example: Overriding Methods

- Suppose we have the following implementation for the getName method in the Person super class,

```
public class Person {
        :
        :
        public String getName(){
        System.out.println("Parent: getName");
        return name;
        }
}
```

Example: Overriding Methods

- To override the getName method of the super class Person in the subclass Student, reimplement the method
- Now, when we invoke the getName method of an object of the subclass Student, the overridden getName method would be called,

```
public class Student extends Person{
      :
      public String getName(){
         System.out.println("Student: getName");
         return name;
         }
      :
}
```

- The output would be,

  Student: getName

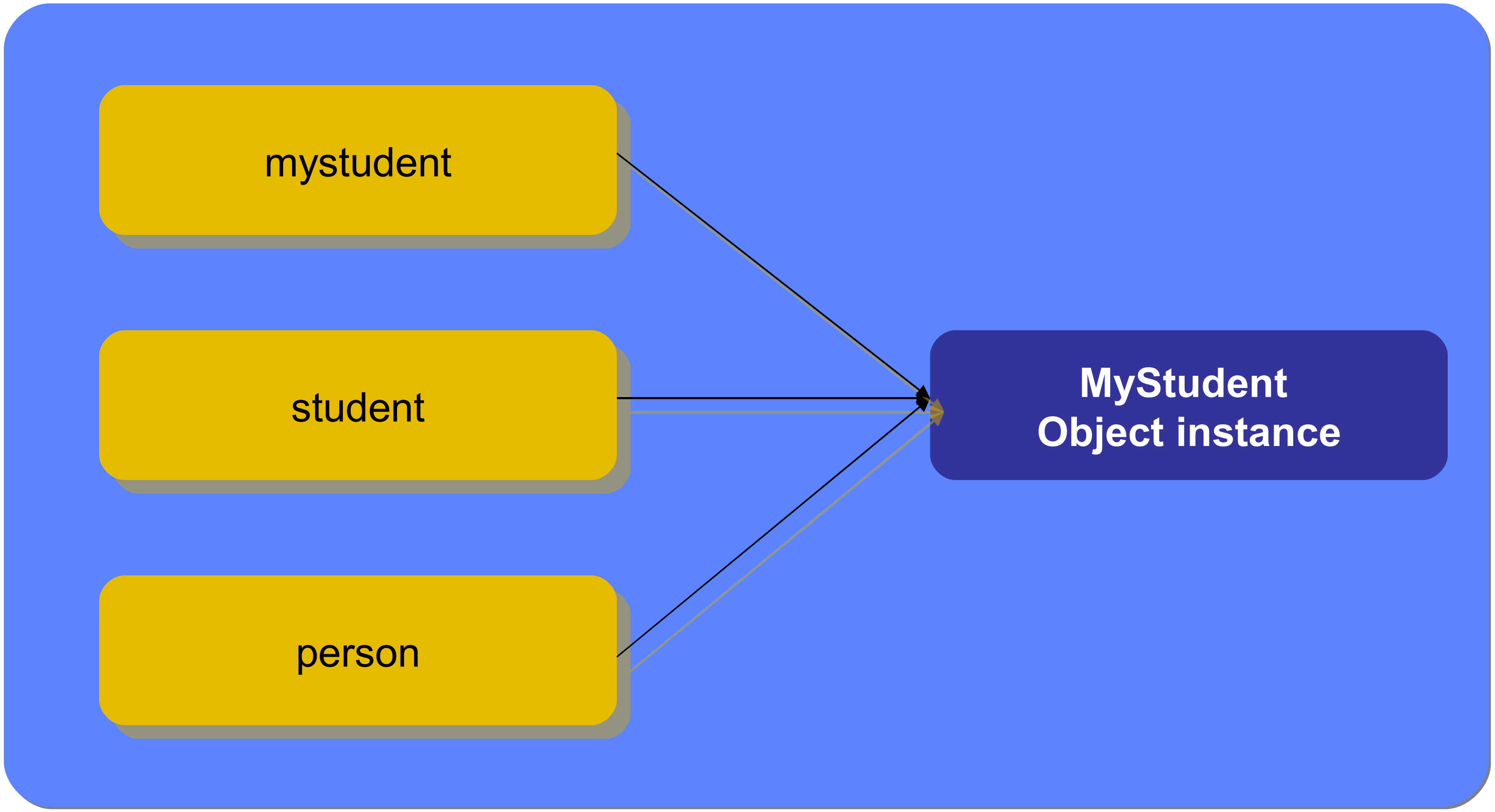
Inheritance: Type casting

Type Casting between Objects

- An object instance of a subclass can be assigned to a variable (reference) of a parent class through implicit type casting
- Example
  - Let's assume Student class is a child class of Person class
  - Let's assume MyStudent class is a child class of Student class

```
MyStudent mystudent = new MyStudent();
Student student = mystudent; // Implicit type casting
Person person = mystudent; // Implicit type casting
```

Type Casting Between Objects



Inheritance:
Final Class & Final Methods

Final Classes

- Final Classes
  - Classes that cannot be extended
  - To declare final classes, we write,

```
public final ClassName{
. . .
}
```

- Example:
  - Other examples of final classes are wrapper classes and Strings.

Final Methods

- Final Methods
  - Methods that cannot be overridden
  - To declare final methods, we write,

```
public final [returnType] [methodName]
  ([parameters]){
. . .
}
```

- Static methods are automatically final.

Example: final Methods

```
public final String getName(){
return name;
}
```

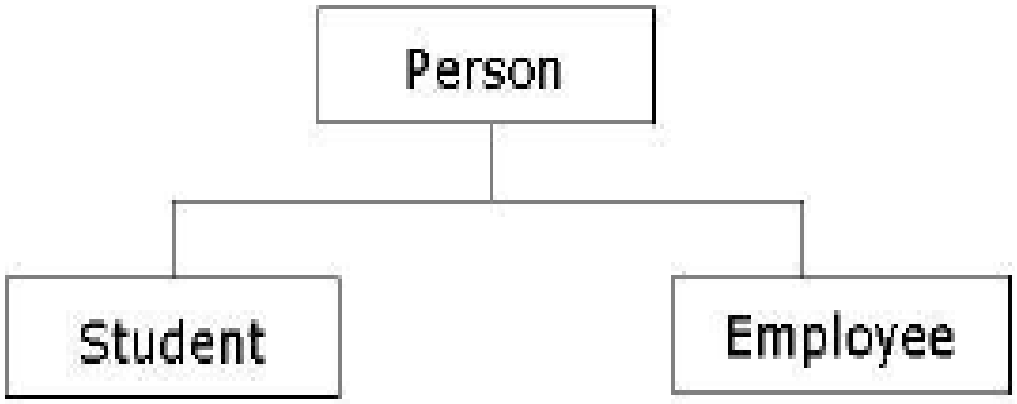
Polymorphism

What is Polymorphism?

- Polymorphism
  - The ability of a reference variable to change behavior according to what object instance it is holding.
  - This allows multiple objects of different subclasses to be treated as objects of a single super class, while automatically selecting the proper methods to apply to a particular object based on the subclass it belongs to.

Example: Polymorphism

- Given the parent class **Person** and the subclass **Student** of the previous examples, we add another subclass of **Person** which is **Employee**.
- Below is the class hierarchy for that,



Example: Polymorphism

- In Java, we can create a reference that is of type super class to an object of its subclass. For example,

```
public void static main( String[] args ) {
Student studentObject = new Student();
Employee employeeObject = new Employee();
Person ref = studentObject; //Person reference points
// to a Student object
}
```

Example: Polymorphism

- Now suppose we have a getName method in our super class Person, and we override this method in both Student and Employee subclass's

```
public class Student extends Person {
	public String getName(){
	System.out.println(“Student Name:” + name);
	return name;
	}
}

public class Employee extends Person {
	public String getName(){
	System.out.println(“Employee Name:” + name);
	return name;
	}
}
```

Polymorphism

- Going back to our main method, when we try to call the `getName` method of the reference Person ref, the `getName` method of the `Student` object will be called.
- Now, if we assign ref to an `Employee` object, the `getName` method of `Employee` will be called.

Example: Polymorphism

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

     Student studentObject = new Student();
     Employee employeeObject = new Employee();

      Person ref = studentObject; //Person ref. points to a
                                // Student object

     // getName() method of Student class is called
            String temp= ref.getName();
      System.out.println( temp );

            ref = employeeObject; //Person ref. points to an
                               // Employee object

            //getName() method of Employee class is called
            String temp = ref.getName();
            System.out.println( temp );
            }
```

Polymorphism

- Another example that illustrates polymorphism is when we try to pass a reference to methods as a parameter
- Suppose we have a static method printInformation that takes in a Person reference as parameter.

```
public static printInformation( Person p ){
                // It will call getName() method of the
                // actual object instance that is passed
    p.getName();
  }
```

Polymorphism

- We can actually pass a reference of type Employee and type Student to the printInformation method as long as it is a subclass of the Person class.

```
public static main( String[] args )
{
  Student studentObject = new Student();
  Employee employeeObject = new
  Employee();
  printInformation( studentObject );
  printInformation( employeeObject );
}
```

Abstract Class

Abstract Classes

- Abstract class
  - a class that contains abstract methods, methods which do not have implementation
  - often appears at the top of an object-oriented programming class hierarchy, defining the broad types of actions possible with objects of all subclasses of the class.
  - An abstract class cannot instantiated. Another class has to provide implementation of abstract methods

Abstract Classes

- Abstract methods
  - methods in the abstract classes that do not have implementation
  - To create an abstract method, just write the method declaration without the body and use the abstract keyword
- For example,

public abstract void someMethod();

Sample Abstract Class

```
public abstract class LivingThing {
    public void breath(){
            System.out.println( "Living Thing breathing...");
            }

    public void eat(){
    System.out.println("Living Thing eating...");
            }

            /**
            * abstract method walk
            * We want this method to be overridden by subclasses of
            * LivingThing
            */
public abstract void walk();
}
```

Abstract Classes

- When a class extends the LivingThing abstract class, it is required to override the abstract method walk(), or else, that subclass will also become an abstract class, and therefore cannot be instantiated.
- For example,

```
public class Human extends LivingThing {
public void walk(){
   System.out.println("Human walks...");
   }
}
```

Coding Guidelines

- Use abstract classes to define broad types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Interfaces

What is an Interface?

- An *interface* is a named collection of method definitions (without implementations). An interface can also declare constants.

- All methods of an interface are abstract methods defines the signatures of a set of methods, without the body (implementation of the methods)

- defines a standard and public way of specifying the behavior of classes.

- allows classes, regardless of their locations in the class hierarchy, to implement common behaviors

- a class implements the interface

Interface Declaration

```
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;
}
```

Declaration of Interface

- A method declaration within an interface is followed by a semicolon (;) because an interface does not provide implementations for the methods declared within it.
- All methods declared in an interface are implicitly public and abstract.
- An interface can contain constant declarations in addition to method declarations.
- All constant values defined in an interface are implicitly *public, static,* and *final*.
- you cannot use *transient, volatile, or synchronized* keywords in a member declaration in an interface.
- Also, you may not use the private and protected specifiers when declaring members of an interface.

Example: Interface

// Note that Interface contains just set of method

// signatures without any implementations

```
interface Customer
{
        String getName();
        int getId();
        int getContact();

}
```

Implementing Interfaces

- An interface defines a protocol of behavior.
- A class that implements an interface adheres to the protocol defined by that interface.
- If a class provides the implementation for an interface, it uses *implements* clause in the class declaration.
- when a class implements an interface it comes to an agreement that it will provide implementations for all the methods
- The methods must be implemented as *public*

Implementing Interfaces

```
class PolicyInfo implements Customer
{
    //Declare instance variables(fields)
    String name;
    int customerId;
    int contactNo;

    PolicyInfo(String name,int id,int contactNo)
    {
          this.name=name;
          customerId=id;
          this.contactNo=contactNo;
    }
```

Implementing Interfaces

```
//MEMBER METHOD OF THE CLASS  Implement the methods
    public String getName()
    {
        return name;
    }

    public int getId()
    {
        return customerId;
    }

    public int getContact()
    {
        return contactNo;
    }
}// end of class
```

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 a method called through one of these references, the correct version will be called based on the actual instance of the interface being referred to.
- 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

Accessing Implementations Through Interface References

```
public class IFcaeDemo {
public static void main(String[] args)
    {

    Customer cif = new PolicyInfo("Chiru",12345,55446677);

    //Display the customer information
    System.out.println("Name         :  "+cif.getName());
    System.out.println("ID              :  "+cif.getId());
    System.out.println("COntact No  :  “ +cif.getContact());

    }
}
```

Why do we use Interfaces?

- To reveal an object's programming interface without revealing its implementation
- To model multiple inheritance which allows a class to have more than one super class
- To have unrelated classes implement similar methods

Interface vs. Abstract Class

- Interface methods have no body
- An interface can only define constants
- Interfaces have no direct inherited relationship with any particular class, they are defined independently

Interface vs. Class

- Common:
  - Interfaces and classes are both types
  - This means that an interface can be used in places where a class can be used
  - For example:

    ```
    PersonInterface pi = new Person();
    Person pc = new Person();
    ```
- Difference:
  - You cannot create an instance from an interface

  For example:

```
PersonInterface pi = new PersonInterface();
  //ERROR!
```

Interface vs. Class

- Common:
  - Interface and Class can both define methods
- Difference:
  - Interface does not have any implementation of the methods

Relationship of an Interface to a Class

- A class can only EXTEND ONE super class, but it can IMPLEMENT MANY interfaces.
- For example:

```
public class Person implements PersonInterface,
WhateverInterface {
//some code here
}
```

Relationship of an Interface to a Class

- Another example:

```
public class ComputerScienceStudent extends Student
implements PersonInterface, AnotherInterface{
//some code here
}
```

Inheritance among Interfaces

- Interfaces are not part of the class hierarchy.
- However, interfaces can have inheritance relationship among themselves

```
public interface PersonInterface {
. . .
}

public interface StudentInterface extends PersonInterface {
. . .
}
```

•One interface can extend more than one interfaces

```
interface Inter3 extends Inter1, Inter2,Inter3 {
        ………..
}
```

Warning! Intrefaces cannot grow

- Once an interface is implemented new methods should not be added to the existing interface
- In such cases
  - Either a new interface is created and the new set of methods are added and then this interface is implemented by the required class
  - Or another interface is created by extending the existing interface and the resultant interface is implemented by the class
- if new methods are added to the existing interface then all the classes implementing this interface becomes abstract .

Interface and Polymorphism

- Interfaces exhibit polymorphism as well, since program may call an interface method, and the proper version of that method will be executed depending on the type of object passed to the interface method call.

Object class – Once again!

- Class Object is the root of the class hierarchy.
- Every class has Object as a superclass.
- All objects, including arrays, implement the methods of this class.
- The class we create also inherit the methods from Object class

Method Summary of Object class

protected Object	**clone()** Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	**getClass**() Returns the runtime class of an object.
int	**hashCode()** Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString()** Returns a string representation of the object.
void	**wait**() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	**wait**(long timeout) Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Summary

- Inheritance (super class, subclass)
- Using the super keyword to access fields and constructors of super classes
- Overriding Methods
- Final Methods and Final Classes
- Polymorphism (Abstract Classes, Interfaces)