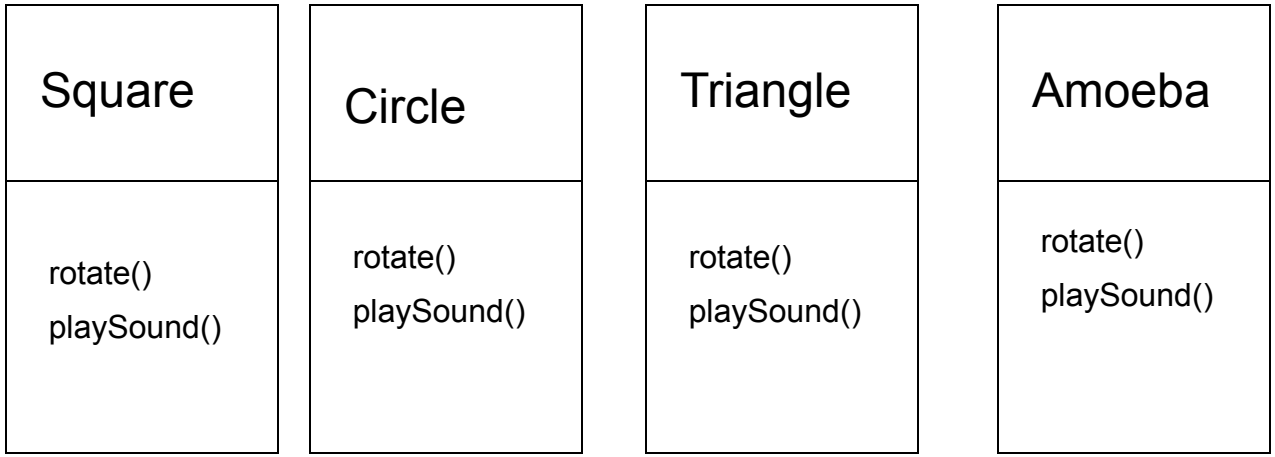
Inheritance

What all four classes have in common?



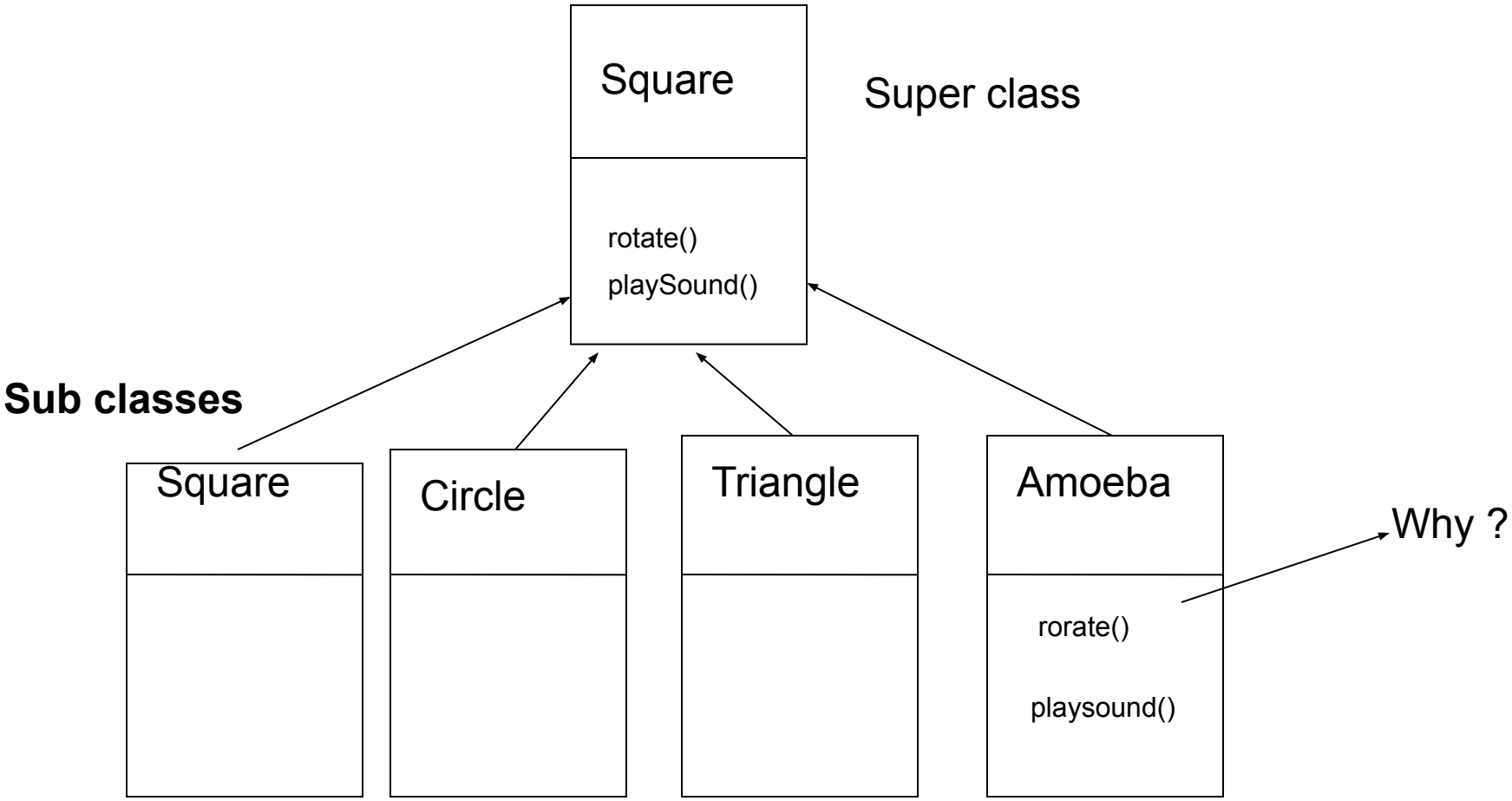
Inheritance

- See all the classes have some common features
- What to do with Common features
- Create a new class , Which will hold all common features
- All other classes will inherit it and use it.

Inheritance is Generalization to specialization.

Creating a new class based on existing class.

Inheritance



Discussion on object relationships

- A is-a relationship
- A has-a relationship

Single inheritance

- In Java this is achieved using “extends” keyword

  public class Manager extends Employee
  {
  …..
  }
- Java allows extensions from only one class
- Single inheritance makes code more reliable
- Using interfaces provides benefits of multiple inheritance without drawbacks

  <<access modifier>> class <<class name>> extends <<super class name>>
  {
  <<declarations and statements>>
  }

Multiple inheritance

- Is it possible in java?

- If yes how?
- Trainer wants to access employee data as well as student data. How to access
- Is the inheritance is only way to access it?
- Will discussed after some time?

Constructors are not inherited

- A subclass inherits all the methods and variables from a parent class
- Constructors are never inherited
- A parent constructor is always called in addition to a child constructor

Polymorphism

- The ability to have many different forms
- An object always has only one form
- A reference variable can refer to objects of different forms

Heterogeneous Collections

- Collections of objects with different class types are called heterogeneous collections

```
Employee[] staff = new Employee[10];
staff [0] = new Manager();
staff [1] = new Engineer();
staff [2] = new Forman();
```

- All objects in Java extend from the Object class
- You can create an array of Object class. Such an array can hold any type of objects. However, such an array cannot hold primitive types

Polymorphic arguments

- Methods can be written to accept "generic" objects

```
public int findSalary (Employee e)
{
        //Do tax calculations
}
```

Elsewhere in the application

```
Manager m = new Manager();
Engineer e = new Engineer();
int managerSalary = findSalary (m);
int engineerSalary = findSalary (e);
```

The instanceof operator

- It is now well understood that you can pass object using references to their parent classes
- In such cases, sometimes you would want to find which class you are dealing with

public class Employee {…..}
public class Manager extends Employee {….}
public class Engineer extends Employee {….}

- Now if you receive an object using reference of type Employee, it might be a Manager or an Engineer. You would obviously want to perform a test.

The instanceof operator

```
public int findDinnerAllowance (Employee e)
{
	if (e instance of Manager)
	{
		return 500;
	}
	else if (e instance of Engineer)
	{
		return 200;
	}
	else
	{
		//Process for some other type of employee
	}
}
```

Casting Objects

- When you case, be sure to use instanceof

```
public double findSalary (Employee e)
{
        if (e instanceof Manager)
        {
                Manager mgr = (Manager) e;
                //Procees salary computations using the mgr object
        }
        else if (e instanceof Engineer)
        {
                //Do the needful processing
        }
}
```

- In the above code, we restored the full functionality of the object by casting

Casting Objects

- Check for proper casting by using the following guidelines
  - Casts up the hierarchy are done implicitly
  - Downward casts must be to a sub-class and checked by the compiler
  - If the compiler allows the casting, the object type is checked at runtime when runtime errors can occur

Access Control – Access modifiers

Modifier	Same Class	Same package	Sub-class	Universe
public	**Yes**	**Yes**	**Yes**	**Yes**
protected	**Yes**	**Yes**	**Yes**	**No**
default	**Yes**	**Yes**	**No**	**No**
private	**Yes**	**No**	**No**	**No**

Method Overloading

- Methods in a class have the same name but different arguments
- Rules for overloaded methods
  - The argument list of the calling statement must differ enough to allow unambiguous determination of proper method to call. Normal widening promotions (e.g. float, double) may be applied. But in some conditions this may cause confusion
  - Return types of the methods may be different, but it is not sufficient for the return type to be the only difference. The argument lists of the overloaded methods must be different

Constructor Overloading

- As with methods, constructors may be overloaded

  public Employee (String name)

  public Employee (String name, int departmentCode)
- Argument lists must differ
- The “this” reference can be used at the first line of the constructor to call another constructor. There may be more initialization code after the this call, but not before

Method Overriding

- A subclass can modify behavior inherited from a parent class
- A sub-class can create a method with different functionality than the parent's method but with the same
  - Name
  - Return type
  - Argument list

Method Overriding

```
public class Employee
{
        protected String name;
        protected double salary;

        public String getDetails()
        {
                return “Name :” + name + “\n” + Salary :” + salary;
        }
}

public class Employee
{
        protected String name;
        protected double salary;
        protected String department;
        public String getDetails()
        {
                return “Name :” + name + “\n” + Salary :” + salary + “\n” + “Manager for :” + department;
        }
}
```

Method Overriding

- Virtual method invocation

  Employee e = new Manager();

  e.getDetails();
- Compile time type and runtime type
- In C++ you get this behavior only if you mark a method virtual. C++ does this to enhance the speed

Method Overriding

- Rules about overridden methods
  - Must have a return type that is identical to the method it overrides
  - Cannot be less accessible than the method it overrides

```
public class Parent
{
        public void doSomething () {…..}
}
public class Child extends Parent
{
        private void doSomething() {…..}
}
public class useBoth ()
{
        public void doOtherThing()
        {
                Parent p1 = new Parent ();
                Parent p2 = new Child ();
                p1.doSomething();
                p2.doSomething();
        }
}
```

  - An overriding method cannot throw different types of exceptions than the method it overrides

The super keyword

- The “super” keyword is used in a class to refer to its superclass
- super can be used to refer to the members of superclass, both data attributes and methods
- The method invoked by using super may not necessarily be in the superclass. It may be further up the hierarchy

The super keyword

```
public class Employee
{
        private String name;
        private double salary;
        public String getDetails()
        {
                return “Name : “ + name + “\nSalary :” + salary;
        }
}
public class Manager extends Employee
{
        private String department;
        public String getDetails()
        {
                return super.getDetails() + “\nDepartment :” + department;
        }
}
```

Invoking Parent Class Constructors

- To invoke a parent class constructor you must place the super keyword in the first line of the constructor
- You can call a specific parent constructor by the arguments that you use in the call to super
- If no “this” or “super” call is used in the constructor, then the compiler adds an implicit call to “super()” which calls the parent default constructor
  - If the parent class does not supply a non-private “default” constructor, then a compiler warning will be issued

Invoking Parent Class Constructors

```
public class Employee
{
        private Employee()
        {
        }
}
```

```
public class Manager extends Employee
{
        public Manager()
        {
        }

}
```

```
public class ConstructorTrial
{
        public static void main(String[] args)
        {
                Manager m = new Manager();
                System.out.println ("Name : " + m.getName());
        }
}
```

Invoking Parent Class Constructors

```
public class Employee
{
        public String name;
        public int salary;

        public Employee(String s, int i)
        {
                name = s;
                salary = i;
        }

        public Employee (String s)
        {
                name = s;
        }
}
```

```
public class Manager extends Employee
{
        public String department;
        public Manager(String name, int salary, String dept)
        {
                super (name, salary);
                department = dept;
        }
        public Manager(String name, String dept)
        {
                super (name);
                department = dept;
        }
        public Manager(String dept)
        {
                department = dept;
        }
}
```

```
public class ConstructorTrial
{
        public static void main(String[] args)
        {
        Manager m = new Manager();
        }
}
```

Constructing and initializing objects – A Reprise

- Memory is allocated for the complete object and default initialization occurs
- The top-level constructor is called and recursively follows the below steps
  1. Bind constructor parameters
  2. If explicit this(), call recursively and then skip to step 5
  3. Call recursively the implicit or explicit super (…), except for Object, because Object has no parent class
  4. Execute explicit instance variable initializers
  5. Execute body of the current constructor

Constructing and initializing objects – A Reprise

```
public class Employee {
    private String name;
    private double salary = 15000.00;
    private Date birthDate;

    public Employee (String n, Date dob)        {
            //implicit super();
            name = n;
            birthDate = dob;
    }

    public Employee (String n)  {
            this (s, null);
    }
}

public class Manager extends Employee  {
    private String department;

    public Manager (String n, String d) {
            super (n);
            department = d;
    }
}
```

Constructing new Manager (“marty”, “Sales”)

0 Basic initialization
- 0.1 Allocate memory for complete Manager object
- 0.2 Initialize all instance variables to their default values (0 or null)

1. Call constructor: Manager (“Nikhil Kapre”, “Sales”)
   - 1.1. Bind constructor parameters: n = “Nikhil Kapre”, d = “Sales”
   - 1.2 There is no explicit this() call
   - 1.3 Call super(n) for Employee (String)
     - 1.3.1 Bind constructor parameters: n = “Nikhil Kapre”
     - 1.3.2 Call this (n, null) for Employee (String, Date)
       - 1.3.2.1 Bind constructor parameters: n = “Nikhil Kapre”, dob = null
       - 1.3.2.2 no explicit this() call
       - 1.3.2.3 Call super() for Object()
         - 1.3.2.3.1 No binding necessary
         - 1.3.2.3.2 No this() call
         - 1.3.2.3.3 No super() call (Object class is the root)
         - 1.3.2.3.4 No explicit variable initialization for object
         - 1.3.2.3.4 No method body to call
       - 1.3.2.4 Initialize explicit Employee variables: salary = 15000.00
       - 1.3.2.5 Execute body: name = “Nikhil Kapre”; date = null;
     - 1.3.3 and 1.3.4 steps skipped
     - 1.3.5 Execute body: No body in Employee (String)
   - 1.4 No explicit initializers for Manager
   - 1.5 Execute body: department = sales

Implications of initializations

```
public class Employee {
    private String name;
    private double salary = 15000.00;
    private Date birthDate;
    private String summary;

    public Employee (String n, Date dob)      {
            //implicit super();
            name = n;
            birthDate = dob;
            summary = getDetails();
    }

    public Employee (String n)  {
            this (s, null);
    }

    public String getDetails () {
            return "Name :" + name + "\nSalary :" +
        salary +
            "\nBirthDate :" + birthDate;
    }
}
```

```
public class Manager extends Employee  {
        private String department;
        public Manager (String n, String d) {
                super (n);
                department = d;
        }

        public String getDetails () {
                return super.getDetails () +
                "\nDepartment :" + department;
        }
}
```

The Object class

- It is the root of all classes in Java
- A class declaration with no “extends” clause, implicitly uses “extends Object”

```
public class Employee
{
    ....
}
```

is equivalent to

```
public class Employee extends Object
{
    ....
}
```

Some significant methods of Object class

- hashCode()
- equals()
- toString()
- wait()
- notify()
- notifyAll()

== operator compared with the equals() method

- The == operator determines if two references are identical to each other (i.e., refer to the same object)
- The equals() method determines if the objects are “equal” but not necessarily identical
- The Object implementation of equals () uses the == operator
- User classes can override the equals method to implement domain specific test for equality
- You should override the hashCode () method if you override the equals method

Example for equals()

```
public class Employee {
        private String name;
        private MyDate birthDate;
        private float salary;

        public Employee (String name, MyDate dob, float salary) {
                this.name = name;
                this.birthDtae = dob;
                this.salary = salary;
        }

        //Define getters and setters for name, birthDate, and salary attributes

        public boolean equals (Object o) {
                if  ((o != null) && (o istanceof Employee)) {
                        Employee e = (Employee) o;
                        if (name.equals (e.name) && birthDate.equals (e.birthDate)) {
                                return true;
                        }
                }
                        return false;
        }
```

toString()

- Converts an object to a String
- Used during string concatenation
- Override the method to provide information about a user-defined object in a readable format
- Primitive types are converted to a String using the wrapper class's toString static method

Wrapper classes

- Java does not look at primitives data types as objects
- The primitive types are maintained more for the sake of efficiency
- Java provides wrapper classes to manipulate primitive data elements as objects
- Each wrapper encapsulates a single primitive value
- Wrapper classes implement immutable objects, i.e. once the primitive value is initialized in them, there is no means to change that value

Wrapper classes

Primitive Data Type	Wrapper class
boolean	**Boolean**
byte	**Byte**
short	**Short**
int	**Integer**
long	**Long**
float	**Float**
double	**Double**
char	**Character**

```
int pInt = 500;
Integer wWint = new Integer (500);
int x = wInt.intValue();
```

```
String s = “221”;
Integer wVal = Integer.valueOf (s);
int x = wVal.intValue();
```

```
String s = “221”;
int x = Integer.valueOf(s).intValue();
```

Advanced Class Features

The “static” keyword

- The static keyword is used as a modifier on variables, methods, and inner classes
- The static keyword declares that the attribute or method is associated with the class as a whole rather than any particular instance of a class
- The static members are often called “class members”, such as “class attributes” or “class methods

The “static” keyword

```
public class Count {
        private int serialNumber;
        private static int counter = 0;

        public static int getTotalCount()
        {
                return counter;
        }

        public count () {
                counter++;
                serialNumber = counter;
        }
}
```

```
public class TestCounter {
        public static void main (String[] args)
        {
                System.out.println (“Counter
        value : “
                        + Count.getTotalCount());

                Count count1 = new Count();
                System.out.println (“Counter
        value : “
                        + Count.getTotalCount());
}
```

The “static” keyword

- Because you can invoke a static method without any instance of the class to which it belongs, there is no “this” value
- The consequence of the above is that a static method cannot access any variables apart from local variables, static attributes, and its parameters
- Non static variables are bound to an instance and can be accessed only through instance references

The “static” keyword

- A static method cannot be overridden
- main() is defined static because the JVM does not create an instance of the class when executing the main method
- Non static variables are bound to an instance and can be accessed only through instance references. So if you have member data, you must create an object to access it.

Static Initializers

- A class can contain code in a static block that does not exist within a method body
- Static block code executes only once when the class is loaded
- A static block is generally used to initialize static variables

Static Initializers

```
public class Count {
        public static int counter;
        static {
                counter = Integer.getInteger("myApp.Count.Counter").intValue();
        }
}
```

```
public class TestStaticInit {
        public static void main (String[] args) {
                System.out.println ("Counter : " + Count.counter);
        }
}
```

Singleton Pattern

- Discuss the need for it
- Allows us to ensure that only one instance of a class can reside
- Allows single point of access to the functionality
- Example, connection pool, controller servlets

Singleton example

```
public class Company {
        private static Company instance = new Company();
        private String name;
        private Vehicle[] fleet;

        public static Company getCompany() {
                return instance;
        }

        private Company()
        {….}
}
```

```
public class FuelNeedsReport {
        public void generateText (PrintStream output) {
                Company c = Company.getCompany();
                //Use company object to retrieve the fleet vehicles
        }
}
```

The final keyword

- You cannot subclass a final class
- You cannot override a final method
- A final variable can be only set once, but that assignment can occur independent of the declaration. Such variables are called “blank final variables”
  - A blank final instance attribute must be set in every constructor
  - A blank final method variable must be set in the method body before being used

Final Classes

- Java allows you to apply final to classes
- Final classes cannot be sub-classed
- An ideal usage scenario is where you don’t want to allow your class to be extended for security reasons
- E.g. String class is declared final

Final Method

- Java allows you to apply final to methods
- Final methods cannot be overridden
- An ideal usage scenario is where you don’t want to allow your method implementations to change to ensure a consistent state of your object
- Methods declared final are sometimes used for optimizations
- When used in situations where class hierarchies exist, using final with methods leads to significant performance enhancements
- Methods marked static or private are automatically marked final because dynamic binding cannot be applied in either case

Final Variables

- Variables marked final are treated as constants
- Any attempt to change a final variable results in a compiler error
- If you mark a reference type variable final, you cannot use it to refer to any other object. You can however change the object's contents

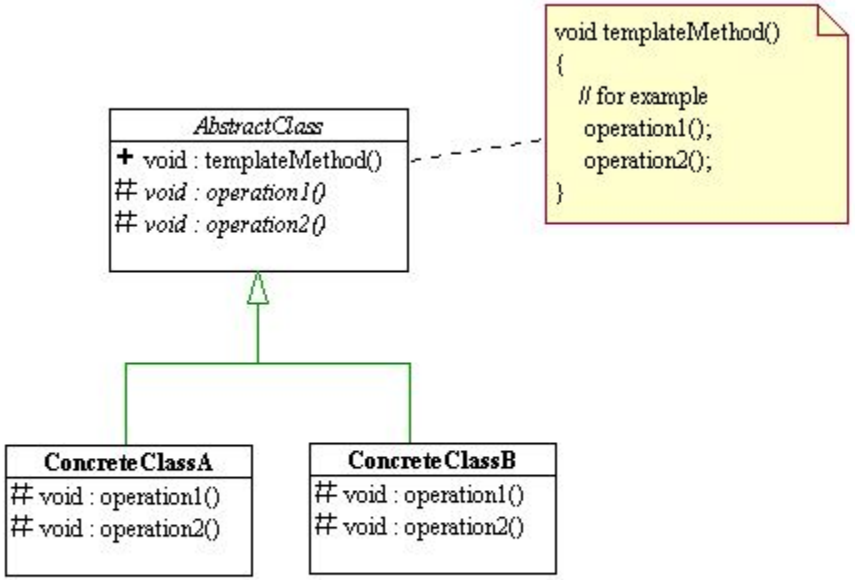
Abstract classes

- Java allows creation of a superclass that declares a method but does not supply any implementation
- Such a method is called abstract method
- Any class with one or more abstract method(s) is called an abstract class
- You cannot create instances of abstract classes
- Abstract classes may have data attributes, concrete methods, and constructors
- It is a good practice to make constructors protected

Abstract classes

- If a sub-class does not provide implementation for abstract methods, it must be declared abstract
- A failure to do so will result in compiler errors

Abstract classes – Template Method Pattern



Abstract classes – Template Method Pattern

- One of the most widely used patterns to setup the outline of an algorithm
- Details are left to specific implementations for later on
- Sub-classes override parts of the algorithm without changing the overall structure
- The invariant behaviour is placed in a template method
- Sub-classes can override abstract methods and provide specifics needed in the respective contexts

Interfaces

- An interface is a contract between client code and the class that implements that interface
- In Java it is identified by the “interface” keyword
- A class can implement many, unrelated interfaces
- Many unrelated classes can implement the same interface

<<access modifier>> class <<class name>> extends <<super class name>>
implements <<interface1>> [, <<interface 2>>]*
{
<<declarations>>*
}

Interfaces

- Concrete classes that implement an interface must implement every method in the interface
- The interface name can be used as a type of reference variable. The usual dynamic binding takes place
- You can use the instanceof operator to determine if an object's class implements an interface

Interfaces

```
public interface IFly {
        public void takeOff ();
        public void land ();
        public void fly ();
}
```

```
public class Airplace implements
        IFly {
        public void takeOff () {
                //Accelerate until lift
        off
                //Raise landing gear
        }
        public void land () {
                //Lower landing gear
                //Decellerate and lower
        flaps
                //Apply wheel brakes
        }
        public void fly () {
                //Keep engine running
        }
}
```

Interfaces

```
public class Bird implements IFly
        {
        public void takeOff () {
                //Make a jump
                //Start flapping wings
        }
        public void land () {
                //Align wings to land
                //Glide until landing
                //Stop with foot
        }
        public void fly () {
                //Keep flapping wings
        }

}
```

```
public class Superman
        implements IFly {
        public void takeOff () { … }
        public void land () { … }
        public void fly () { … }
        public void jumpBuilding() {
        … }
}
```

Uses of Interfaces

- Declaring methods that one or more classes are expected to implement
- Determining an object's programming interface without revealing the actual body of the class
- Capturing similarities between unrelated classes without forcing a class relationship
- Simulating multiple inheritance by declaring a class that implements several interfaces

Difference Between Interface and Abstract Class

Abstract Class	*Interface*
must not be instantiated	must not be instantiated
may contain static and final data	variables are implicitly static and final
abstract class can have non-abstract methods but, abstract method should be inside an abstract class	methods are implicitly abstract. Therefore, all methods should be implemented in the subclass which implements it. no method implementation strictly in the interface.
abstract method should not contain any of these keywords - private, final, static, native, synchronized [pfsns]	methods in interface should not contain any of these keywords - protected, private, final, static, native, synchronized [ppfsns]
methods are not implicitly public	methods are implicitly public even if not specified (be careful when overriding)
is an incomplete class	specification or prescription for behavior
can extend only one parent class	can implement several interfaces atonce
can have constructors (should contain body)	interfaces can't have constructors

Inner Classes

- Added to JDK 1.1
- Allow class definition to be placed inside another class definition
- Group classes that logically belong together
- Inner classes have access to the enclosing class's scope

Inner Classes

```
public class Outer1 {
        private int size;

        public class Inner {
                public void doStuff () {
                        size++;         //Inner class has access to the size variable
                }
        }

        public void testTheInner () {
                Inner i = new Inner ();
                i.doStuff();
        }
```

Inner Classes

```
public class Outer2 {
        private int size;

        public class Inner {
                public void doStuff () {
                        size++;         //Inner class has access to the size variable
                }
        }
}

public class TestInner () {
        public static void main (String[] args) {
                Outer2 outer = new Outer2 ();

                //Declaration of the Inner object happens relative to an Outer
                Outer2.Inner inner = outer.new Inner ();
                inner.doStuff ();
        }
}
```

Inner Classes – Disambiguate variables

```
public class Outer3 {
        private int size;

        public class Inner {
                private int size;
                public void doStuff (int size) {
                        size++;                 //Local parameter
                        this.size++;            //The Inner object attribute
                        Outer3.this.size++ //The outer object attribute
                }
        }
}
```

Inner Classes – Within the scope of a method

```
public class Outer4 {
        private int size = 5;
        public Object makeTheInner (int localVar) {
                final int finalLocalVar = 6;

                //Declare a class within a method
                class Inner {
                        public String toString () {
                                return (“Inner size = ” + size +
                                //”LocalVar = “ + localVar +              //Illegal access
                                “finalLocalVar = “ + finalLocalVar);
                                );
                        }
                return Inner ();
                }
        }
}
```

Properties of Inner Classes

- Inner class name should differ from the enclosing class name
- You can use the class name only within the defined scope
- The inner class declared within a method can only access final local variables
- Inner classes can use both class and instance variables of instance classes
- Inner classes can be declared abstract
- Inner classes can have any access modifier
- Inner classes can act as an interface implemented by another class

Properties of Inner Classes

- Inner classes that are declared static automatically become top level classes
- Inner classes cannot declare any static members, only top level classes can
- An inner class wanting to use static members must be declared static

Variables in interface

```
import java.util.*;
interface sharedconstants
{
        int NO=0;
        int YES=1;
}
class Question implements sharedconstants
{
        Random r=new Random();
        int ask();
        {
                int prob=(int)(100*r.nextDouble());
                if(prob<30){return(NO);}
                else {return(YES);}
        }
}
```

Interface can Extended

- One interface can extend another

```
interface A
{
}
interface B extends A
{}
```

Assignment

- Define a interface ‘Stack’
- Having following abstract methods:push,pop
- Write a class for implementing above methods,keep a check for stack overflow and underflow