# Example: using *super* with constructors

```
class Animal
{
    Animal()
    {
        System.out.println("An instance of Animal class is constructed");
    }
}

class Cat extends Animal
{
    Cat()
    {
        super();
        System.out.println("An instance of Cat class is constructed");
    }
}

public class Test
{
    public static void main(String args[])
    {
        Cat cat = new Cat();
    }
}
```

```
An instance of Animal class is constructed
An instance of Cat class is constructed
```

A call to *super()* has to be the first line in the child class constructor (compile error otherwise)

If not explicit call to the parent class constructor is made then Java compiler inserts an implicit call

- Constructor chaining is invoking constructors of every parent class until you get to the base class. If no explicit constructor calls are made then compiler automatically inserts constructor chaining
- In this example, if no constructor without any arguments exists in the parent class a compile-time error will be displayed

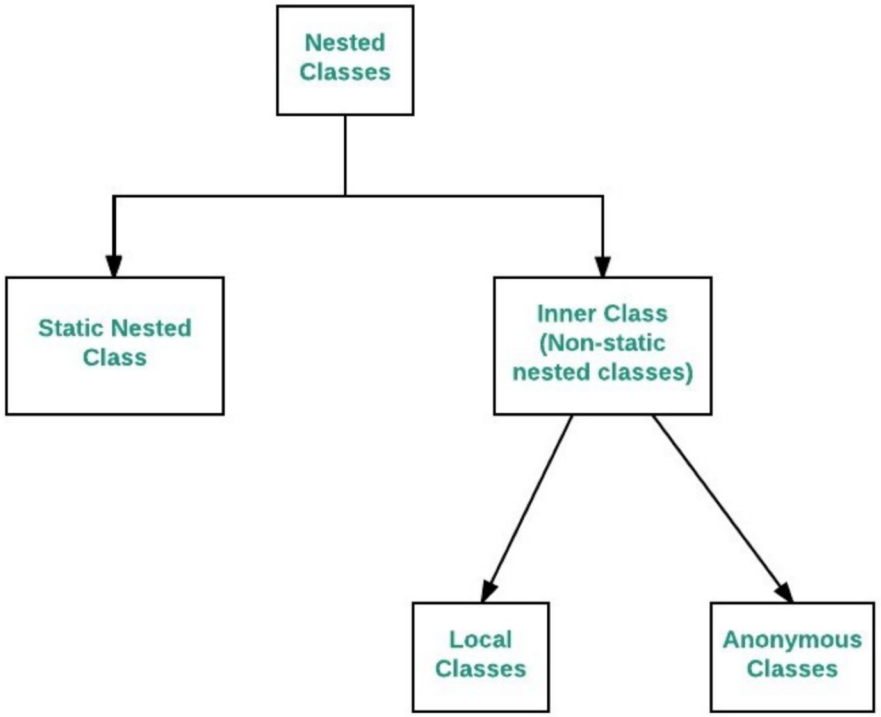
# Nested classes in Java

- In Java we can define one class inside of another class
- Allow creating and using a class in-place if it will not be used anywhere else
- Scope of the nested class is bounded by the scope of the enclosing class
- Nested class has access to all members, even private, of the enclosing class
- Nested class is considered to be a member of the enclosing class

# Example: basic structure of a nested class

```
class MyClass
{
...
    class MyNestedClass
    {
        ...
    }
}
```

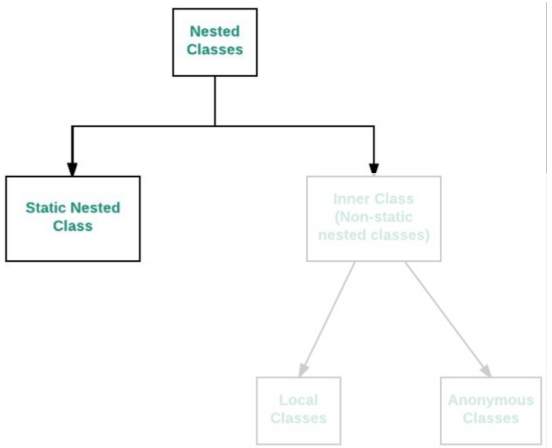
# Two types of nested classes




https://www.geeksforgeeks.org

# Static nested classes

- If a nested class declared static then it is called “static nested class”




https://www.geeksforgeeks.org

# Example: nested static class

```java
class OuterClass {
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private static member
    private static int outer_private = 30;

    // static nested class
    static class StaticNestedClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);

            // can access display private static member of outer class
            System.out.println("outer_private = " + outer_private);

            // The following statement will give compilation error
            // as static nested class cannot directly access non-static members
            // System.out.println("outer_y = " + outer_y);

        }
    }
}
```

```java
// Driver class
public class Test {
    public static void main(String[] args)
    {
        // accessing a static nested class
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

        nestedObject.display();

    }
}
```

Nested static class is a static member of the enclosing class so can be declared without instantiating the outer class

Output:

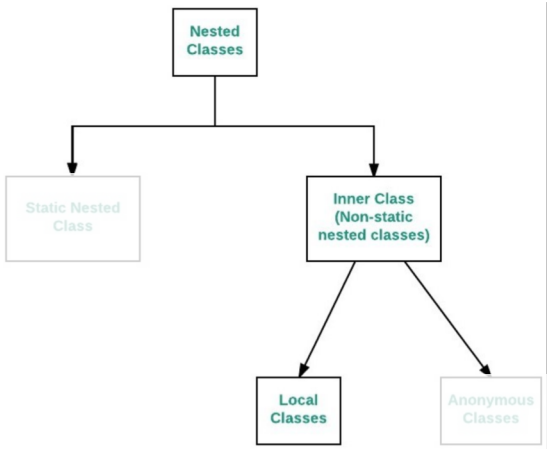
```
outer_x = 10
outer_private = 30
```

This is a static nested class, so only has access to static members of the outer class

https://www.geeksforgeeks.org

# Inner nested classes

- Non-static inner classes can be declared only after the outer class has been instantiated

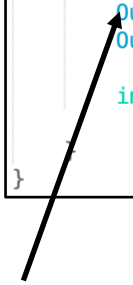



https://www.geeksforgeeks.org

# Example: nested non-static inner class

```java
class OuterClass
{
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private int outer_private = 30;

    // inner class
    class InnerClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);

            // can also access non-static member of outer class
            System.out.println("outer_y = " + outer_y);

            // can also access private member of outer class
            System.out.println("outer_private = " + outer_private);

        }
    }
}
```

```java
// Driver class
public class InnerClassDemo
{
    public static void main(String[] args)
    {
        // accessing an inner class
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();

        innerObject.display();

    }
}
```



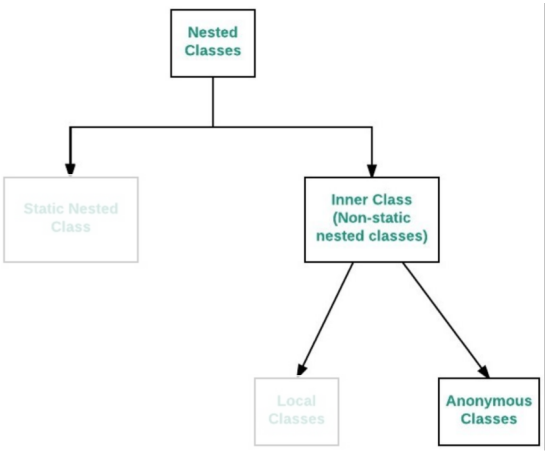
We first must instantiate the outer class and then we can instantiate the inner class

Output:

```
outer_x = 10
outer_y = 20
outer_private = 30
```

https://www.geeksforgeeks.org

# Anonymous classes




https://www.geeksforgeeks.org

- Inner class without a name
- A single object is created
  - No reuse
- Declare and instantiate class at the same time
- From the syntax perspective, anonymous class is an expression
  - Similar to invoking a constructor but there is a class declaration that comes with it

```
Test t = new Test()
{
   // data members and methods
   public void test_method()
   {
      ........
      ........
    }
};
```

# What can we declare in anonymous classes?

- Anonymous classes are useful when making an instance of an object with some “extras”
- Can declare
  - Fields
  - Methods
  - Instance initializers
  - Local classes
- Cannot declare a constructor in anonymous classes
  - Can pass in constructor arguments

# Example: simple anonymous class

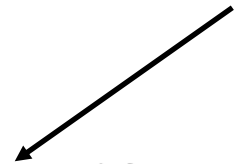
```
abstract class Animal
{
    abstract void sound();
}

public class Test
{
    public static void main(String args[])
    {
        Animal a = new Animal(){
            @Override
            void sound(){System.out.println("I am an animal");}
        };
        a.sound();
    }
}
```

- An anonymous class is created by compiler with a name internal to compiler, extends Animal
- An instance of a class with name “a” is created, its type is Animal

# Example: giving ”extra” behavior to an instance of the Animal class without explicitly creating a new class that extends Animal

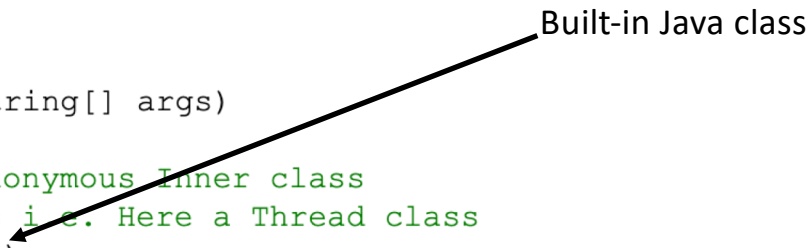
```
abstract class Animal
{
    abstract void sound();
}

public class Test
{
    public static void main(String args[])
    {
        Animal a = new Animal(){
            @Override
            void sound(){System.out.println("I am actually a cat");}
        };
        a.sound();
    }
}
```

Cat behavior without creating a Cat class



# Example: anonymous class that extends a class

```
class MyThread
{
    public static void main(String[] args)
    {
        //Here we are using Anonymous Inner class
        //that extends a class i.e. Here a Thread class
        Thread t = new Thread()
        {
            public void run()
            {
                System.out.println("Child Thread");
            }
        };
        t.start();
        System.out.println("Main Thread");
    }
}
```




https://www.geeksforgeeks.org

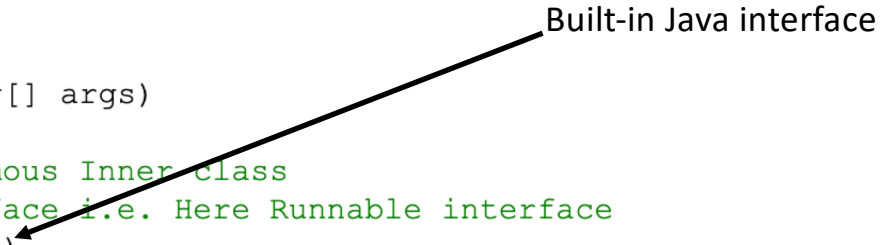
# Example: anonymous inner class that implements interface

```
interface Age
{
    int x = 21;
    void getAge();
}
class AnonymousDemo
{
    public static void main(String[] args) {

        // Myclass is hidden inner class of Age interface
        // whose name is not written but an object to it
        // is created.
        Age oj1 = new Age() {
            @Override
            public void getAge() {
                 // printing  age
                System.out.print("Age is "+x);
            }
        };
        oj1.getAge();
    }
}
```

https://www.geeksforgeeks.org

# Example 2: anonymous inner class that implements interface

```
class MyThread
{
    public static void main(String[] args)
    {
        //Here we are using Anonymous Inner class
        //that implements a interface i.e. Here Runnable interface
        Runnable r = new Runnable()
        {
            public void run()
            {
                System.out.println("Child Thread");
            }
        };
        Thread t = new Thread(r);
        t.start();
        System.out.println("Main Thread");
    }
}
```




https://www.geeksforgeeks.org

# Scope of the anonymous class

- Has access to the members of the enclosing class
- If there is a declaration of a variable in the anonymous class with the same name as the enclosing class, then this declaration *shadows* the declaration in enclosing class
  - Variable cannot be referred to by its name alone

# Example: scope of anonymous class variables

```
public class ShadowTest {

    public int x = 0;

    class FirstLevel {

        public int x = 1;

        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
        }
    }

    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

Output:

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

https://docs.oracle.com

# Why use nested classes?

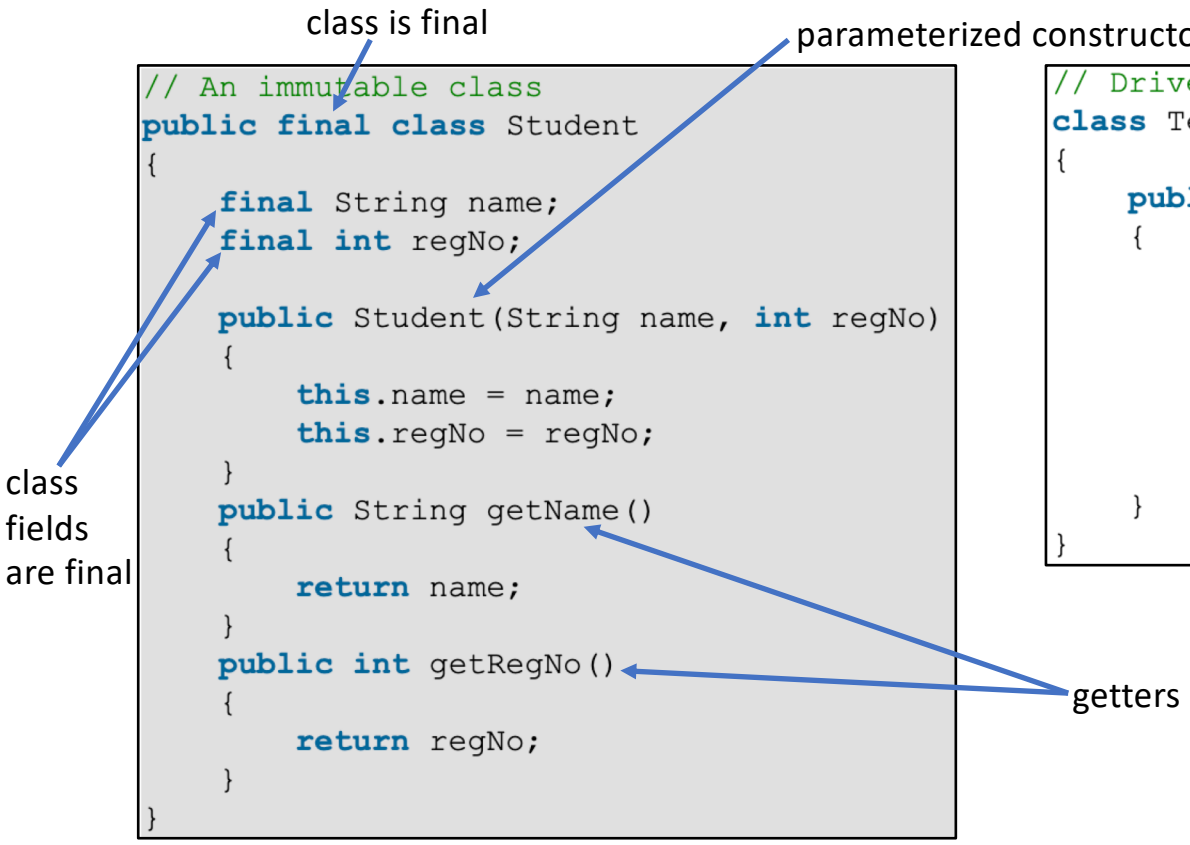
- Lets you group classes that are only used once
  - Encapsulation – OOD principle that promotes hiding of data in class implementation
  - Better control over data access to the members of a nested class
- More readable code
- Easier to maintain code
  - Avoid adding unnecessary *.java* files to the project
- UI event listeners

# Immutable classes

- Once an instance of the class is instantiated its contents cannot be changed
- Java does not provide a direct way to implement immutable classes
  - Class has to be declared as final
    - No inheritance is allowed
  - All fields of the class should be final
  - Parameterized constructors
  - No setters
- Initialized by its constructor

```
final class ImmutableClass
{
    private final int var1;
    private final String var2;

    ImmutableClass(int v1, String v2){
        this.var1 = v1;
        this.var2 = v2;
    }
}
```

# Example: immutable class




```java
// An immutable class
public final class Student
{
    final String name;
    final int regNo;

    public Student(String name, int regNo)
    {
        this.name = name;
        this.regNo = regNo;
    }
    public String getName()
    {
        return name;
    }
    public int getRegNo()
    {
        return regNo;
    }
}
```

```java
// Driver class
class Test
{
    public static void main(String args[])
    {
        Student s = new Student("ABC", 101);
        System.out.println(s.getName());
        System.out.println(s.getRegNo());

        // Uncommenting below line causes error
        // s.regNo = 102;
    }
}
```

Output:

```
ABC
101
```

https://www.geeksforgeeks.org; Abhishree Shetty

# Why use immutable classes?

- Immutable objects remain in one state
  - Thread safe (multi-threaded programming)
  - No synchronization issues
- Immutable objects are easier to design, implement and use
- Internal state of the system is consistent between runs
- References to immutable objects are cached, which can provide faster operations
- In practice, ability to use immutable objects has been shown to provide better scalability

# Built-in Java *String* class is immutable

- *java.lang.String* class
- *String myString = "abc";*

Example:

```
public class Test
{
    public static void main(String args[])
    {
        String myString = "abc";
        System.out.println("myString = "+myString);

        myString = "ced";
        System.out.println("myString = "+myString);
    }
}
```

Output:

```
myString = abc
myString = ced
```

How is this possible if String object is immutable?

# Built-in Java *String* class is immutable

- *java.lang.String* class
- *String myString = "abc";*

Example:

```
public class Test
{
    public static void main(String args[])
    {
        String myString = "abc";
        System.out.println("myString = "+myString);

        myString = "ced";
        System.out.println("myString = "+myString);
    }
}
```

Output:

```
myString = abc
myString = ced
```

How is this possible if String object is immutable?

- You are not changing the value of the original string
- A new String object is actually created
- A new memory location is allocated and the reference is changed to that new location
- Immutable means you cannot change an object itself, not the reference

# Interfaces

- Like classes, interfaces have attributes and methods
- Classes can implement interfaces
- Methods declared in interfaces are by default abstract
- Blueprint for a class
  - Specifies what a class should and should not implement but does not provide implementation itself
  - Every class that implements a given interface must implement methods declared by the interface
    - If not, then the class must be declared abstract
- Classes “inherit” from interfaces by using a keyword *implements*

```
interface Printable {
    void print();
}
```

# Example: Vehicle interface

```
import java.io.*;

interface Vehicle {

    // all are the abstract methods.
    void changeGear(int a);
    void speedUp(int a);
    void applyBrakes(int a);
}
```

https://www.geeksforgeeks.org

```
class Bicycle implements Vehicle{

    int speed;
    int gear;

     // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
              + " gear: " + gear);
    }
}
```

https://www.geeksforgeeks.org

```
class Bike implements Vehicle {

    int speed;
    int gear;

    // to change gear
    @Override
    public void changeGear(int newGear){

        gear = newGear;
    }

    // to increase speed
    @Override
    public void speedUp(int increment){

        speed = speed + increment;
    }

    // to decrease speed
    @Override
    public void applyBrakes(int decrement){

        speed = speed - decrement;
    }

    public void printStates() {
         System.out.println("speed: " + speed
             + " gear: " + gear);
    }

}
```

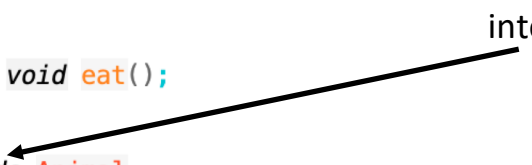
https://www.geeksforgeeks.org

# Why use interfaces?

- Abstraction
  - OOD principle
- Achieve multiple inheritance
  - Java does not allow inheritance from multiple parent classes
  - Classes can implement more than one interface
    - Interfaces are listed using a comma after the keyword *implements*

# Example: interface extends another interface

```java
interface Animal
{
    public abstract void eat();
}

interface Pet extends Animal
{
    public abstract void sound();
}

class Cat implements Pet
{
    public void sound(){System.out.println("Meow");}
    public void eat(){System.out.println("Cat is eating");}
}

public class test
{
    public static void main(String args[])
    {
        Cat c1 = new Cat();
        c1.sound();
        c1.eat();
    }
}
```




Output:

```
Meow
Cat is eating
```

# Let’s take quiz #2

- In-class quiz

# Example: nested static class

```java
class OuterClass {
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private static member
    private static int outer_private = 30;

    // static nested class
    static class StaticNestedClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);

            // can access display private static member of outer class
            System.out.println("outer_private = " + outer_private);

            // The following statement will give compilation error
            // as static nested class cannot directly access non-static members
            // System.out.println("outer_y = " + outer_y);

        }
    }
}
```

```java
// Driver class
public class Test {
    public static void main(String[] args)
    {
        // accessing a static nested class
        OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

        nestedObject.display();

    }
}
```

Nested static class is a static member of the enclosing class so can be declared without instantiating the outer class

Output:

```
outer_x = 10
outer_private = 30
```

This is a static nested class, so only has access to static members of the outer class

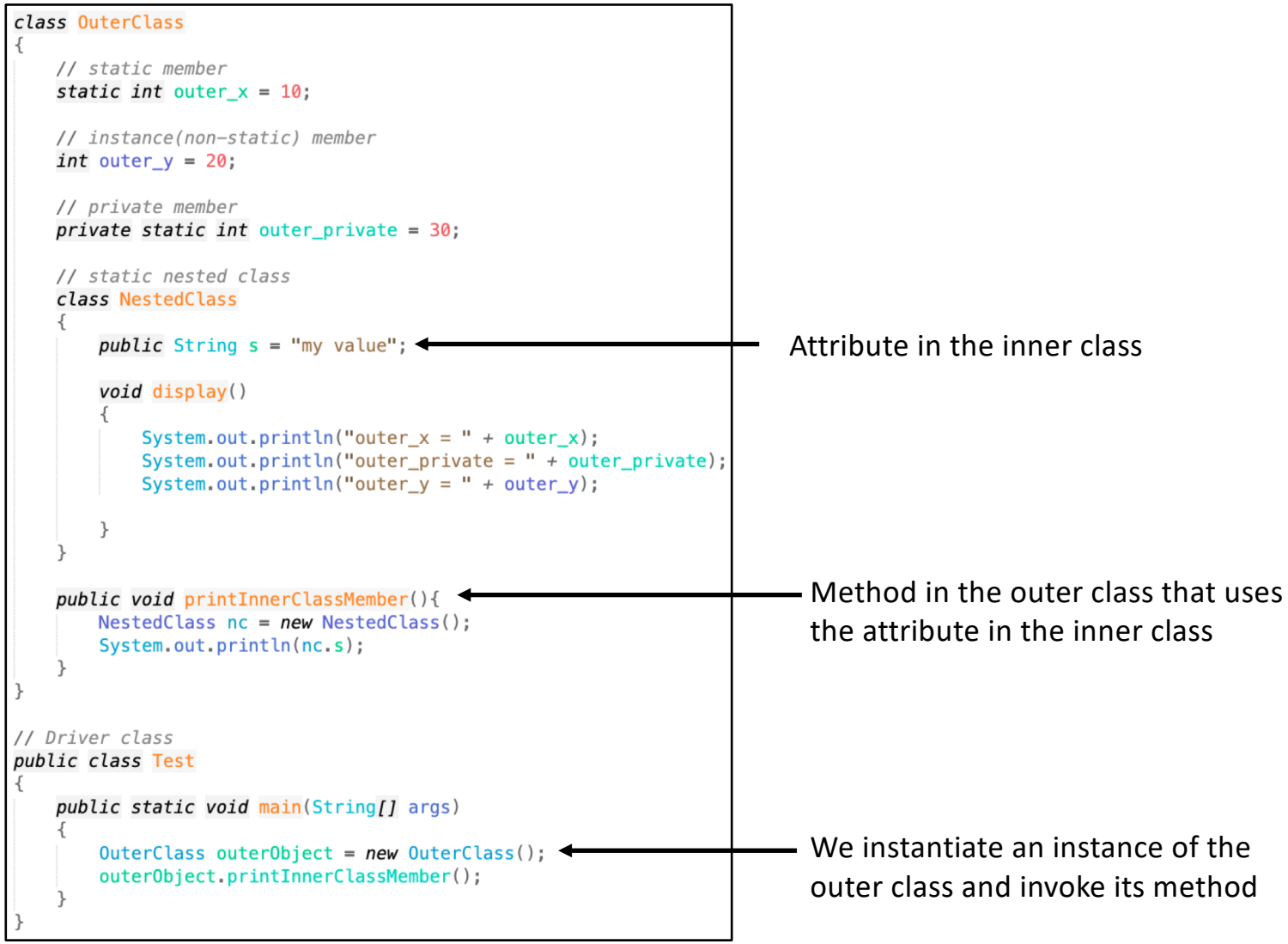
https://www.geeksforgeeks.org

```java
class OuterClass
{
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

    // static nested class
    static class StaticNestedClass
    {
        void display()
        {
            // can access static member of outer class
            System.out.println("outer_x = " + outer_x);

            // can access display private static member of outer class
            System.out.println("outer_private = " + outer_private);

            // The following statement will give compilation error
            // as static nested class cannot directly access non-static members
            // System.out.println("outer_y = " + outer_y);

        }
    }
}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        // accessing a static nested class
        //OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
        //nestedObject.display();

        OuterClass.StaticNestedClass.display();

    }
}
```

```
Test.java:12187: error: non-static method display() cannot be referenced from a static context
        OuterClass.StaticNestedClass.display();
                                    ^
1 error
```

# Can the outer class access members of an inner class?

- It depends




Attribute in the inner class

Method in the outer class that uses the attribute in the inner class

We instantiate an instance of the outer class and invoke its method

```java
class OuterClass
{
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

    // static nested class
    class NestedClass
    {
        private String s = "my value";

        void display()
        {
            System.out.println("outer_x = " + outer_x);
            System.out.println("outer_private = " + outer_private);
            System.out.println("outer_y = " + outer_y);

        }
    }

    public void printInnerClassMember(){
        NestedClass nc = new NestedClass();
        System.out.println(nc.s);
    }
}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        OuterClass outerObject = new OuterClass();
        outerObject.printInnerClassMember();
    }
}
```



Even if the attribute in the inner class is private it still works

```java
class OuterClass
{
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

    // static nested class
    static class NestedClass
    {
        private String s = "my value";

        void display()
        {
            System.out.println("outer_x = " + outer_x);
            System.out.println("outer_private = " + outer_private);
            //System.out.println("outer_y = " + outer_y);

        }
    }

    public void printInnerClassMember(){
        NestedClass nc = new NestedClass();
        System.out.println(nc.s);
    }
}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        OuterClass outerObject = new OuterClass();
        outerObject.printInnerClassMember();
    }
}
```

Now we have a static inner class

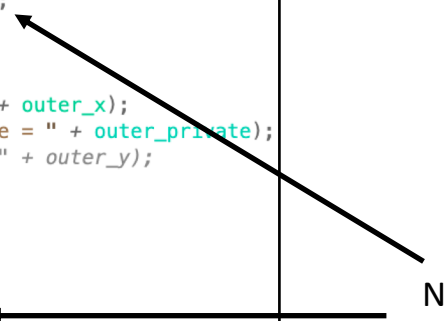
Still works but we instantiate an instance of the static inner class; this is because our attribute is not static

```java
class OuterClass
{
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

    // static nested class
    static class NestedClass
    {
        private String s = "my value";

        void display()
        {
            System.out.println("outer_x = " + outer_x);
            System.out.println("outer_private = " + outer_private);
            //System.out.println("outer_y = " + outer_y);

        }
    }

    public void printInnerClassMember(){
        System.out.println(NestedClass.s);
    }
}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        OuterClass outerObject = new OuterClass();
        outerObject.printInnerClassMember();
    }
}
```

```
Test.java:12174: error: non-static variable s cannot be referenced from a static context
        System.out.println(NestedClass.s);
                                       ^
1 error
```

Won't work because our attribute is not static

```java
class OuterClass
{
    // static member
    static int outer_x = 10;

    // instance(non-static) member
    int outer_y = 20;

    // private member
    private static int outer_private = 30;

    // static nested class
    static class NestedClass
    {
        private static String s = "my value";

        void display()
        {
            System.out.println("outer_x = " + outer_x);
            System.out.println("outer_private = " + outer_private);
            //System.out.println("outer_y = " + outer_y);

        }
    }

    public void printInnerClassMember(){
        System.out.println(NestedClass.s);
    }
}

// Driver class
public class Test
{
    public static void main(String[] args)
    {
        OuterClass outerObject = new OuterClass();
        outerObject.printInnerClassMember();
    }
}
```



Now this works because the attribute is static

# Interface vs. abstract class

- Methods
  - Interface can have only abstract methods
  - Abstract class can have abstract and non-abstract methods
- Variables
  - All variables declared in interface are final
  - Abstract class can have final and non-final variables
- Access modifiers
  - Members of an interface are public by default
  - Abstract class can have members of any access level
- Non-access modifiers
  - Interface: static and final
  - Abstract class: final, non-final, static and non-static
- Inheritance
  - Interface can implement other interface(s)
  - Abstract class can extend another class or implement other interface(s)

# Example: Shape abstract class

```
import java.io.*;

abstract class Shape
{
    private String objectName = " ";

    Shape(String name)
    {
        this.objectName = name;
    }

    public void move(int x, int y)
    {
        System.out.println(this.objectName + " " + "has been moved to"
                                + " x = " + x + " and y = " + y);
    }

    abstract public double area();
    abstract public void draw();
}
```

Inspired by https://www.geeksforgeeks.org

Both classes inherit from abstract class Shape

```java
class Rectangle extends Shape
{
    private int length, width;

    Rectangle(int length, int width, String name)
    {
        super(name);
        this.length = length;
        this.width = width;
    }

    @Override
    public void draw()
    {
        System.out.println("Rectangle has been drawn ");
    }

    @Override
    public double area()
    {
        return (double)(length*width);
    }
}
```

Overriding abstract method

```java
class Circle extends Shape
{

    private double pi = 3.14;
    private int radius;

    Circle(int radius, String name)
    {

        super(name);
        this.radius = radius;
    }

    @Override
    public void draw()
    {

        System.out.println("Circle has been drawn ");
    }

    @Override
    public double area()
    {
        return (double)((pi*radius*radius)/2);
    }
}
```

*Example: Shape abstract class continued*

```java
public class Test {
    public static void main(String args[])
    {
        Shape rect = new Rectangle(2,3, "Rectangle");
        System.out.println("Area of rectangle: " + rect.area());
        rect.move(1,2);

        System.out.println(" ");

        Shape circle = new Circle(2, "Cicle");
        System.out.println("Area of circle: " + circle.area());
        circle.move(2,4);
    }
}
```

```
Area of rectangle: 6.0
Rectangle has been moved to x = 1 and y = 2

Area of circle: 6.28
Cicle has been moved to x = 2 and y = 4
```

*Example: Shape abstract class continued*

# Example: Shape interface

```
import java.io.*;

interface Shape
{
    abstract public void move(int x, int y);
    abstract public double area();
    abstract public void draw();
}
```

Both classes implement interface Shape

```
class Rectangle implements Shape
{
    private int length, width;

    Rectangle(int length, int width)
    {
        this.length = length;
        this.width = width;
    }

    public void draw()
    {
        System.out.println("Rectangle has been drawn ");
    }

    public double area()
    {
        return (double)(length*width);
    }

    public void move(int x, int y)
    {
        System.out.println("Rectangle has been moved to"
                           + " x = " + x + " and y = " + y);
    }
}
```

```
class Circle implements Shape
{

    private double pi = 3.14;
    private int radius;

    Circle(int radius)
    {
        this.radius = radius;
    }

    public void draw()
    {

        System.out.println("Circle has been drawn ");
    }

    public double area()
    {
        return (double)((pi*radius*radius)/2);
    }

    public void move(int x, int y)
    {
        System.out.println("Circle has been moved to"
                           + " x = " + x + " and y = " + y);
    }
}
```

*Example: Shape interface continued*

No changes to the class that uses Rectangle and Circle classes

```java
public class Test {
    public static void main(String args[])
    {
        Shape rect = new Rectangle(2,3);
        System.out.println("Area of rectangle: " + rect.area());
        rect.move(1,2);

        System.out.println(" ");

        Shape circle = new Circle(2);
        System.out.println("Area of circle: " + circle.area());
        circle.move(2,4);
    }
}
```

```
Area of rectangle: 6.0
Rectangle has been moved to x = 1 and y = 2

Area of circle: 6.28
Cicle has been moved to x = 2 and y = 4
```

*Example: Shape interface continued*

# Marker interface

- Special interfaces in Java without method declarations
- These are not “contracts”
  - Allow attaching a particular characteristic to a class that implements a marker interface
  - E.g. a class that implements *Clonable* interface is available for cloning but the implementation details are up to the class
- The idea is to provide a way to say “yes I am something”
- Typical interfaces specify functionality while marker interfaces specify object type

# Cloneable built-in interface

- A member of *java.lang* package
- Class that implements *Cloneable* interface is available for field-for-field cloning into a copy of an instance of this class
  - Using *clone()* method
- Classes that implement *Cloneable* must implement method *clone()*

```
public interface Clonable
{
  // nothing here
}
```

# Example: Cloneable interface, class A

```java
// Java program to illustrate Cloneable interface
import java.lang.Cloneable;

// By implementing Cloneable interface
// we make sure that instances of class A
// can be cloned.
class A implements Cloneable
{
    int i;
    String s;

    // A class constructor
    public A(int i,String s)
    {
        this.i = i;
        this.s = s;
    }

    // Overriding clone() method
    // by simply calling Object class
    // clone() method.
    @Override
    protected Object clone()
    throws CloneNotSupportedException
    {
        return super.clone();
    }
}
```

```java
public class Test
{
    public static void main(String[] args)
        throws CloneNotSupportedException
    {
        A a = new A(20, "GeeksForGeeks");

        // cloning 'a' and holding
        // new cloned object reference in b

        // down-casting as clone() return type is Object
        A b = (A)a.clone();

        System.out.println(b.i);
        System.out.println(b.s);
    }
}
```

https://www.geeksforgeeks.org/

# Example: Cloneable interface, class DogName

```
public class DogName implements Cloneable {

   private String dname;

   public DogName(String dname) {
        this.dname = dname;
   }

   public String getName() {
        return dname;
   }
   // Overriding clone() method of Object class
   public Object clone()throws CloneNotSupportedException{
        return (DogName)super.clone();
   }

   public static void main(String[] args) {
        DogName obj1 = new DogName("Tommy");
        try {
                DogName obj2 = (DogName) obj1.clone();
                System.out.println(obj2.getName());
        } catch (CloneNotSupportedException e) {
                e.printStackTrace();
        }
   }
}
```

https://beginnersbook.com/2015/01/cloneable-interface-in-java-object-cloning/

# Serializable built-in interface

- A member of *java.io* package
- Gives ability to any instance of the class that implements Serializable to be available for saving its state to a stream (serialization)
  - A stream is usually output to a file
- ObjectInputStream and ObjectOutputStream classes implement *writeObject()* and *readObject()* methods

```
public interface Serializable
{
  // nothing here
}
```

# Example: Serializable interface

```java
// Java program to illustrate Serializable interface
import java.io.*;

// By implementing Serializable interface
// we make sure that state of instances of class A
// can be saved in a file.
class A implements Serializable
{
    int i;
    String s;

    // A class constructor
    public A(int i,String s)
    {
        this.i = i;
        this.s = s;
    }
}
```

```java
public class Test
{
    public static void main(String[] args)
      throws IOException, ClassNotFoundException
    {
        A a = new A(20,"GeeksForGeeks");

        // Serializing 'a'
        FileOutputStream fos = new FileOutputStream("xyz.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(a);

        // De-serializing 'a'
        FileInputStream fis = new FileInputStream("xyz.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        A b = (A)ois.readObject();//down-casting object

        System.out.println(b.i+" "+b.s);

        // closing streams
        oos.close();
        ois.close();
    }
}
```

https://www.geeksforgeeks.org/

# You can create your own marker interface

- Write your own marker interface the same way you would write any other interface
  - No method declarations
- Test if the object is the type defined by interface
  - Use *instanceof* keyword

```
public interface MyMarkerInterface {

}

public void myMethod(Object object) throws InvalidEntityException {
    if(!(object instanceof MyMarkerInterface)) {
        throw new InvalidEntityException("Invalid object type");
    }
    ...
}
```

# Hybrid interfaces

- No restrictions in Java to have hybrid interfaces, those that specify both behavior and type
- Generally not a good idea, things might become messy

# Main differences between a class and interface

| Class | Interface |
|---|---|
| Keyword *class* | Keyword *interface* |
| Inheritance with *extends* | Inheritance with *implements* |
| Can extend only one class and implement any number of interfaces | Interface can extend any number of interfaces but cannot implement any interfaces |
| Can be instantiated | Cannot be instantiated |
| Can have constructors | No constructors |
| Methods are defined (accept for abstract classes) | Methods are always abstract |
| Access modifiers for members: private, public, protected | Access modifiers for members: only public |
| Any non-access modifiers for the variables | All variables are static and final |

# To recap

- Classes
  - Basic unit of OOP
  - Follow guidelines for good class design
  - Nested classes (static nested classes, non-static inner classes, anonymous classes)
  - Immutable classes
  - Abstract classes
- Interfaces
  - Difference between interfaces and abstract classes
  - Marker interfaces
  - Important build-in interfaces
  - Hybrid interfaces
- Other important concepts
  - Access modifiers allow control over access to data and methods
    - Packages are a useful tool for controlling access
  - Non-access modifiers add greater control over your program structure and use cases
  - Classes can be instantiated while interfaces cannot be instantiated

# Concluding remarks

- Classes and interfaces are units of OOD and important to know for OOP
- Both classes and interfaces have their own roles and purposes
- Best way to learn OOP is to practice