MAHARISHI UNIVERSITY of MANAGEMENT

Engaging the Managing Intelligence of Nature

Computer Science Department

CS390 Foundamental Programming Practices (FPP)

Professor Paul Corazza

Lecture 6:
Nested Classes

Wholeness of the Lesson

Nested classes allow classes to play the roles of instance variable, static variable and local variable, providing more expressive power to the Java language. Likewise, it is the hidden, unmanifest dynamics of consciousness that are responsible for the huge variety of expressions in the manifest world.

Outline of Topics

1. **Definitions of Four Types of Nested and Inner Class**
2. Most Commonly Used Nested Classes: Member and Static
3. Examples in Context of Sorting: The Comparator interface
4. Local and Anonymous Inner Classes and Sorting
5. Using Lambda Expressions in Place of Anonymous Inner Classes

Definition and Types of Nested Classes (see `lesson6_inner`)

- A class is a *nested class* if it is defined *inside* another class. (Note: This is different from having multiple classes defined in the same file.) A nested class is an *inner class* if it has access to all members (variables and methods) of its enclosing class (described below).

- Four kind of nested classes:
  - member
  - static
  - local
  - anonymous

- Of these, member, local, and anonymous are all called *inner* classes.

- An alternative, but equivalent, definition of *inner class* is "any non-static nested class"

- Sometimes in books you will see the term "static inner class" – this is simply loose language for static nested class.

- It is possible to make inner classes `private` and also `static` (see below) – these keywords cannot be used with ordinary classes.

Main Point

Classes are the fundamental concept in Java – programs are built from classes. With nested classes, Java makes it possible for this fundamental construct to play the roles of instance variable (member inner classes), static variable (static nested classes), and local variable (local inner classes). Likewise, in the unfoldment of creation, pure intelligence assumes the role of creative intelligence – in all of creation we find pure intelligence in the guise of individual expressions, individual existences, assuming diversified roles.

Outline of Topics

1. Definitions of Four Types of Nested and Inner Class
2. **Most Commonly Used Nested Classes: Member and Static**
3. Examples in Context of Sorting: The Comparator interface
4. Local and Anonymous Inner Classes and Sorting
5. Using Lambda Expressions in Place of Anonymous Inner Classes

Example of a Member Inner Class

```
public class MyClass {
   private String s = "hello";
   public static void main(String[] args){
      new MyClass();
   }
   MyClass() {
      MyInnerClass myInner = new MyInnerClass();
      System.out.println(myInner.intval);
      myInner.innerMethod();
   }
   private class MyInnerClass {
      private int intval = 3;
      private void innerMethod(){
         System.out.println(s);
      }
   }
}
```

```
//Output:                          [See package lesson6_inner]
```

```
3
hello
```

Member Inner Class Syntax and Rules

- Member inner classes, like other members of the class, can be declared public or private, or may have package level access or may be *protected* . In the example, the inner class has private level access.

- Member inner classes have access to all fields and methods of the enclosing class, including private fields and methods. No explicit reference to an enclosing class instance is needed.

  In the example, notice how the variable `s` in the enclosing class is accessed by an inner class method.

- Likewise, the outer class can access private variables and methods in the inner class, but only with reference to an inner class instance that has already been created. In the example, this is done by the enclosing class's constructor.

- Like ordinary classes, when a member inner class is instantiated, it has an implicit parameter `'this'`. The `'this'` of the enclosing class is accessible from within the inner class . The `'this'` of the inner class is accessible from within itself, but *not* from the enclosing class.

Example

```
Class MyOuterClass {
    MyInnerClass inner;
    private String param;
    MyOuterClass(String param) {
        inner = new MyInnerClass("innerStr");
        this.param = param; // the outer class version of this
    }
    void outerMethod() {
        System.out.println(inner.innerParam);
        inner.innerMethod();
        //String t = inner.this.innerParam; //compiler error
    }
    class MyInnerClass{
        private String innerParam;
        MyInnerClass(String innerParam) {
            //the inner class version of 'this'
            this.innerParam = innerParam;
        }
        void innerMethod() {
            //accessing enclosing class's version of this
            System.out.println(MyOuterClass.this.param);
            //same as the following
            System.out.println(param);
        }
  }
  public static void main(String[] args) {
    (new MyOuterClass("outerStr")).outerMethod();
  }
}
```

```
//OUTPUT:
innerStr
outerSt
outerStr
```

- To access the methods and variables of a member inner class, it is necessary to explicitly instantiate it – it is *not* instantiated automatically when the enclosing class is instantiated.

```
public class MyClass {
   private String s = "hello";
   MyInnerClass inner;
   public static void main(String[] args){
       new MyClass();
   }
   MyClass() {
       System.out.println(inner.anInt);//NullPointerException
       inner = new MyInnerClass();
       System.out.println(inner.anInt); //OK
   }
   class MyInnerClass{
       private int anInt = 3;
       void innerMethod(){
           System.out.println(s);
       }
   }
}
```

- Until jdk 16, member inner classes were not allowed to contain static variables or methods. As of jdk 16, this restriction is no longer valid (see https://openjdk.java.net/jeps/395).

- If the member inner class is sufficiently accessible (i.e., not private), it can be instantiated by a class other than the enclosing class, as long as an instance of the enclosing class has already been created.

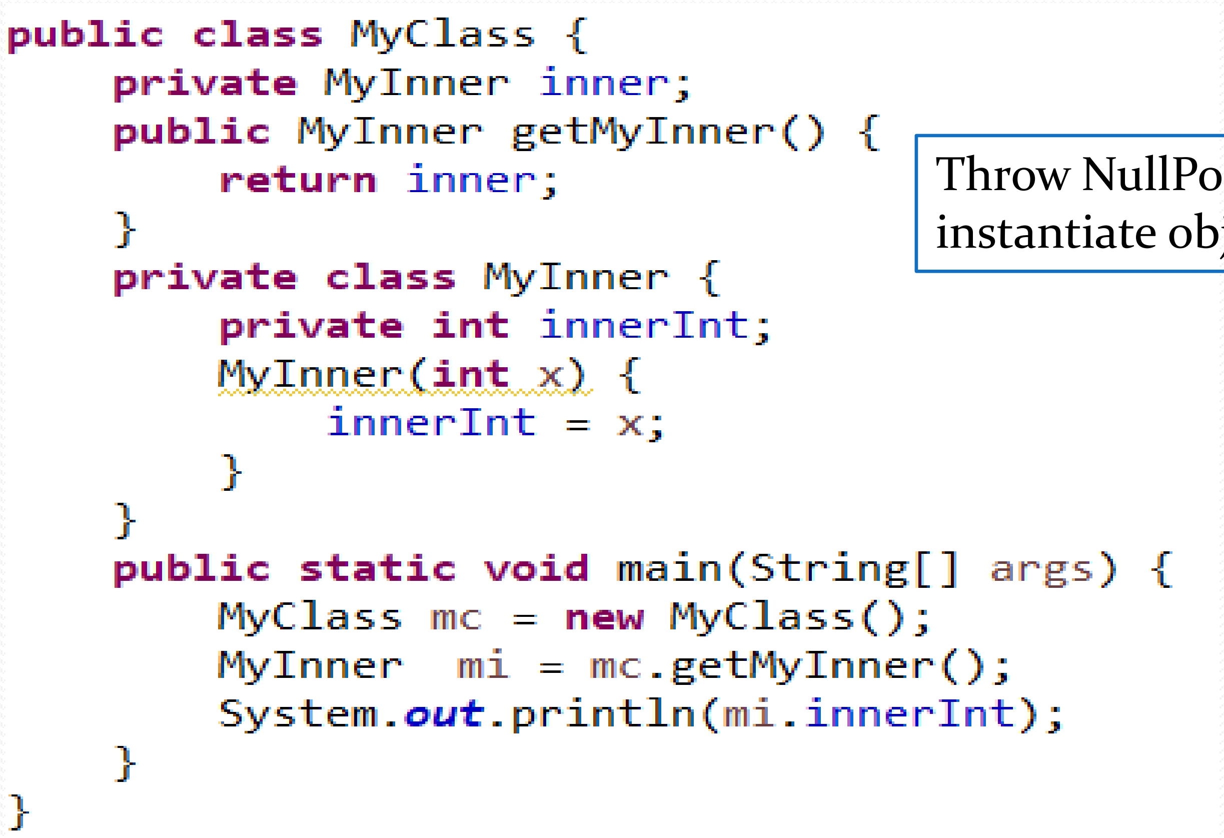
Example:

```
class ClassA {
     class InnerClassA {
     }
}

class ClassB {
    ClassB()  {
    ClassA a = new ClassA();
    ClassA.InnerClassA innerA = a.new InnerClassA(); //ok

    }
}
class ClassC {
    ClassC()  {
    ClassA.InnerClassA innerA =
       new ClassA.InnerClassA();  //illegal, 'new' requires an
                                    //enclosing instance
    }
```

- **Best Practice.** A member inner class is typically a small specialized "assistant" that is exclusively owned by its enclosing class. Consequently, it is not good practice to access a member inner class from outside the enclosing class, since this undermines the overall purpose of this type of nested class.

Exercise 5.1

In each class, what happens? Is there a compiler error? If not, what happens when the main method is run? Runtime error? Successful execution? If successful, what is printed to the console?



```java
public class MyClass {
    private MyInner inner;
    public MyInner getMyInner() {
        return inner;
    }
    private class MyInner {
        private int innerInt;
        MyInner(int x) {
            innerInt = x;
        }
    }
    public static void main(String[] args) {
        MyClass mc = new MyClass();
        MyInner  mi = mc.getMyInner();
        System.out.println(mi.innerInt);
    }
}
```

Throw NullPointerException, Not instantiate object for MyInner

Main Point

Inner classes – a special kind of nested class – have access to the private members of their enclosing class. The most commonly used kind of inner class is a *member* inner class. Likewise, when individual awareness is awake to its fully expanded, self-referral state, the memory of its ultimate nature becomes lively.

Static Nested Classes

- If a nested class is defined using the `static` keyword, it becomes a static nested class.
- Static nested classes do not have access to instance variables and methods of the enclosing class. A static nested class is in effect a top level class that has been "packaged" differently; it has the same access to the enclosing class variables and methods as another class located in the same package.

```
public class Main {
   public int i = 4;
   public int getInt() {
     return 3;
   }
   static class NestedClass {
     public void innerMethod() {
       int j = i;          //compiler error
       int k = getInt(); //compiler error
     }
   }
}
[See package lesson6_inner]
```

- It is possible to declare static variables and methods in a static nested class; however, static nested classes may also have instance variables and methods.

- As with member inner classes, the enclosing class of a static nested class has access to the nested class's private variables and methods, with reference to an instance.

- Unless a static nested class is declared private, other classes can instantiate it, but the syntax is different from that for member inner classes. For example:

```
MyClass.MyStaticNestedClass cl =
                new MyClass.MyStaticNestedClass();
```

- A static nested class may not be defined inside an instance inner class. (Recall that static variables and methods cannot be declared inside a member inner class either.)

```
public class MyClass {
    private String s = "hello";
    public static void main(String[] args){
        new MyClass();
    }
    MyClass() {
        //access static methods in the usual way
        MyStaticNestedClass.myStaticMethod();

        //access instance methods in the usual way too
        //except that now private methods are also accessible
        MyStaticNestedClass cl = new MyStaticNestedClass();
        cl.myOtherMethod();

        //as with inner classes, private instance vbles are accessible
        int y = cl.x;
    }
    static class MyStaticNestedClass {
        private int x = 0;
        static void myStaticMethod() {
            String t = s; //compiler error -- no access
        }
        private void myOtherMethod() {
        }
    }
}
class AnotherClass {
    public static void main(String[] args){
        MyClass.MyStaticNestedClass cl = new MyClass.MyStaticNestedClass(); //OK
        MyClass m = new MyClass();
        //the following is illegal-- compiler error
        MyClass.MyStaticNestedClass cl2 = m.new MyStaticNestedClass();
    }
} // [See package lesson6_inner]
```

- **Best Practice.** Static nested classes should be thought of as ordinary (“top-level” or “first class”) classes that are “privately packaged”. Usually not accessed from outside, but it’s not necessarily bad practice to do so since static nested classes are “top level” classes.
  - The book gives an example of a `Pair` class that is defined within another class `ArrayAlg`; could make `Pair` an ordinary (externally defined) class, but making it static nested class controls the namespace – if another `Pair` class exists in the application, there will be no conflict.
  - In the Java API, `LinkedList.Entry` and `HashMap.Entry` are examples of static nested classes.

Outline of Topics

1. Definitions of Four Types of Nested and Inner Class
2. Most Commonly Used Nested Classes: Member and Static
3. **Examples in Context of Sorting: The Comparator interface**
4. Local and Anonymous Inner Classes and Sorting
5. Using Lambda Expressions in Place of Anonymous Inner Classes

Application: Sorting

- Nested classes are useful in many situations. We illustrate one application to sorting objects. Lesson 6 will introduce another application in development of user interfaces with Swing

The `Comparator` Interface

Problem: Suppose you have an array `arr` of `Person` objects and you want to sort the array in this way:

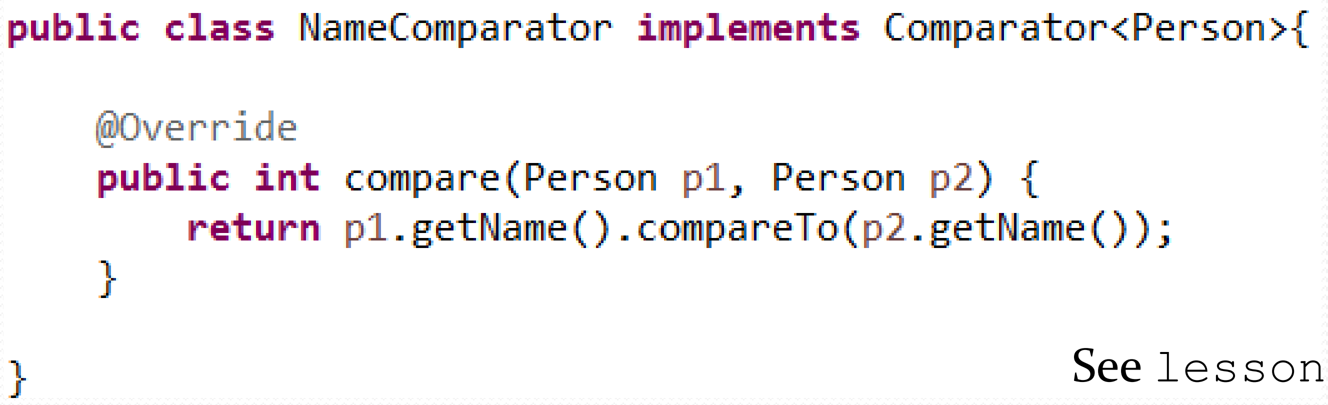
```
        Arrays.sort(arr)
```

The compiler will complain because there is no natural ordering on `Person`, and `Person` does not (in this case) implement `Comparable`.

Solution: Implement a `Comparator` for `Person`. (Another solution would be to have `Person` implement `Comparable`.)

[See `lesson6.comparators` for several demos.]

- Declare person1 to be "less than" person2 if the name of person1 comes before the name of person2 :

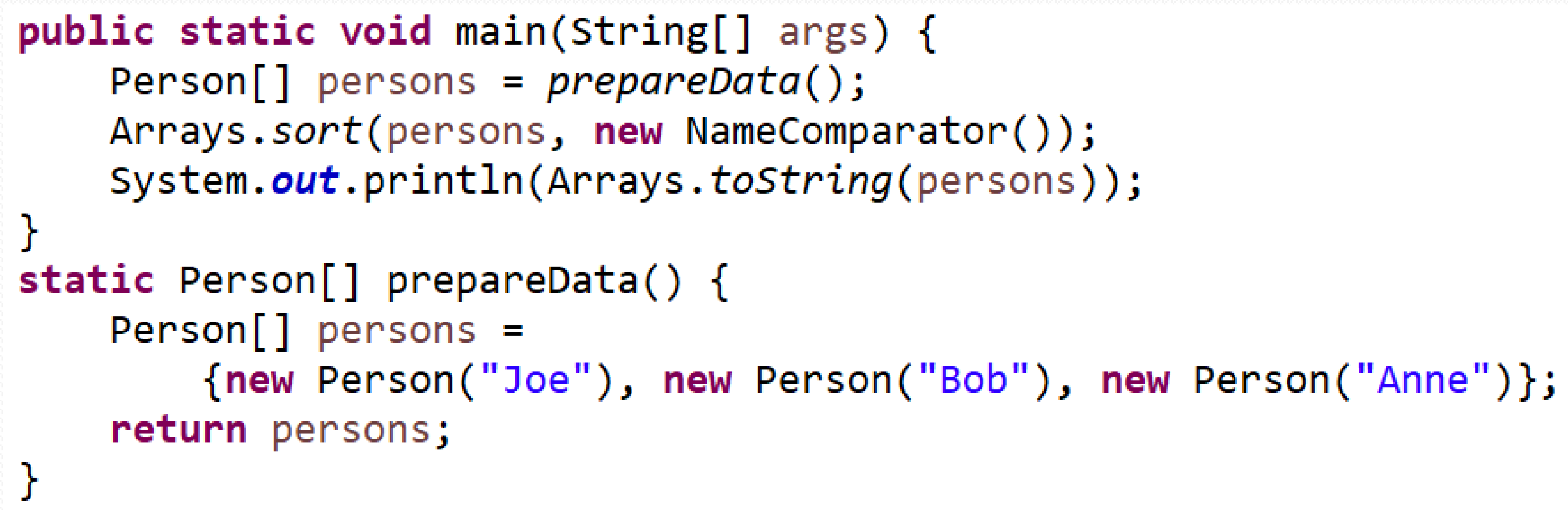


```
public class NameComparator implements Comparator<Person>{

    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }

}
```

See `lesson6_innersort1`

- Then you can sort an array of `Persons` if you also pass in an instance of the `NameComparator`:

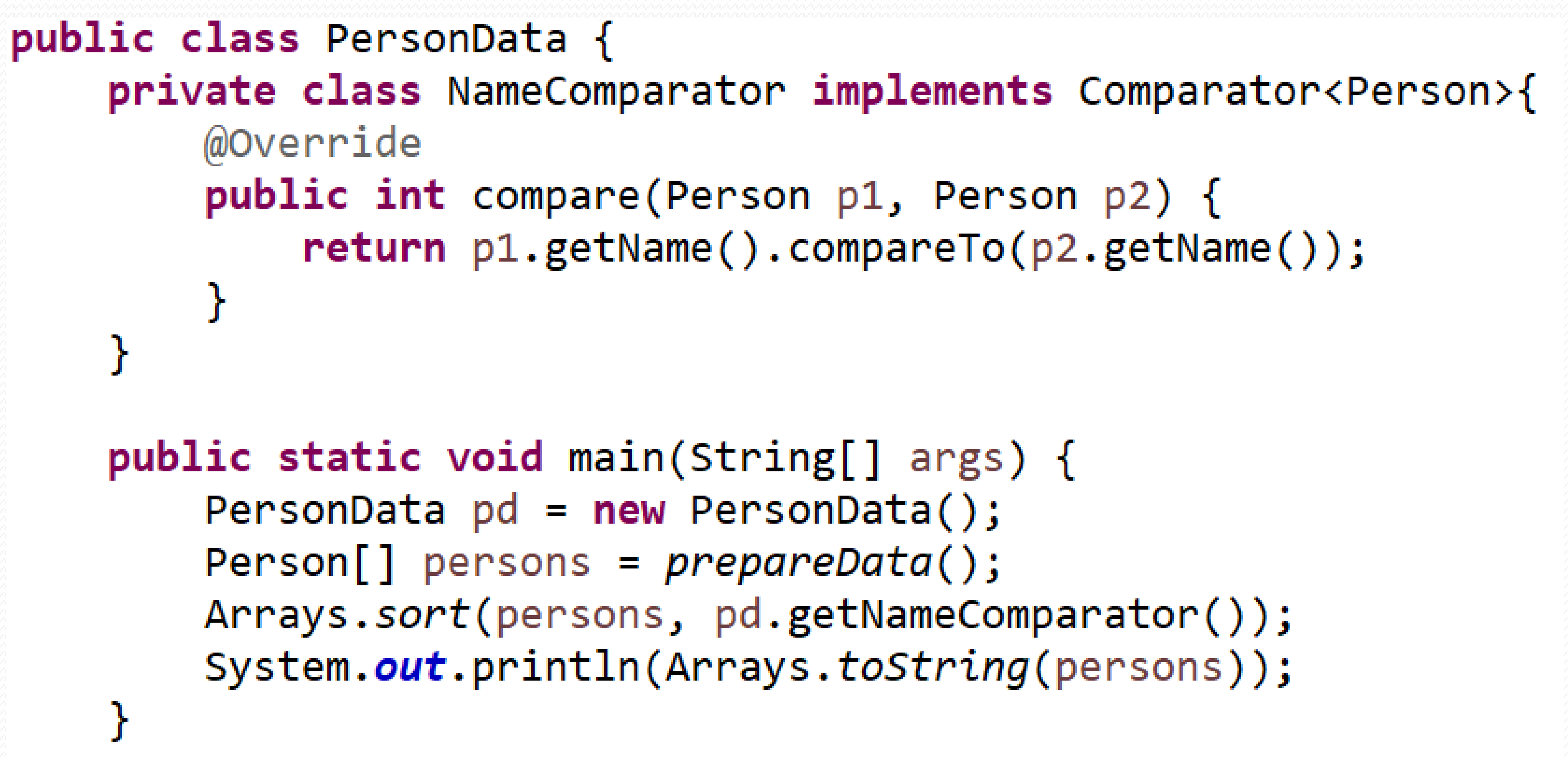
//from class PersonData



```
public static void main(String[] args) {
    Person[] persons = prepareData();
    Arrays.sort(persons, new NameComparator());
    System.out.println(Arrays.toString(persons));
}
static Person[] prepareData() {
    Person[] persons =
        {new Person("Joe"), new Person("Bob"), new Person("Anne")};
    return persons;
}
```

Implementing a Comparator as a Member Inner Class

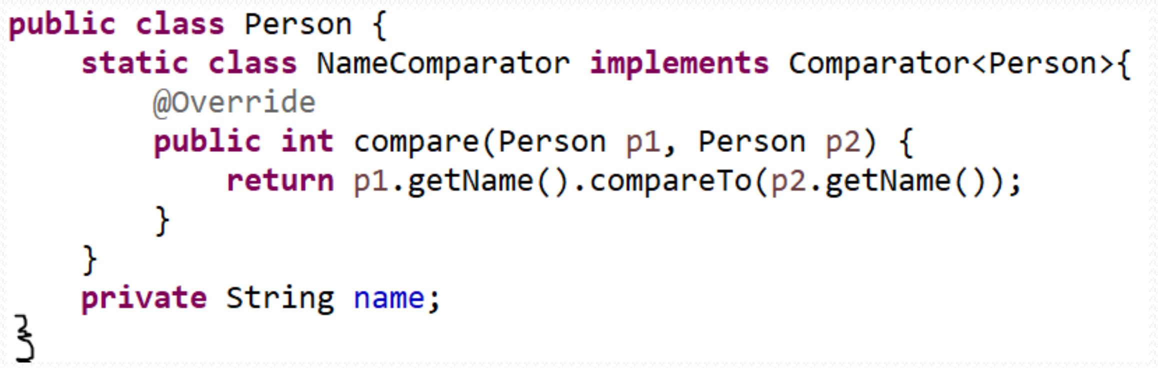
- Can implement NameComparator as a member inner class of PersonData since it is a naturally seen as a support class for PersonData. (See lesson6_innersort2)



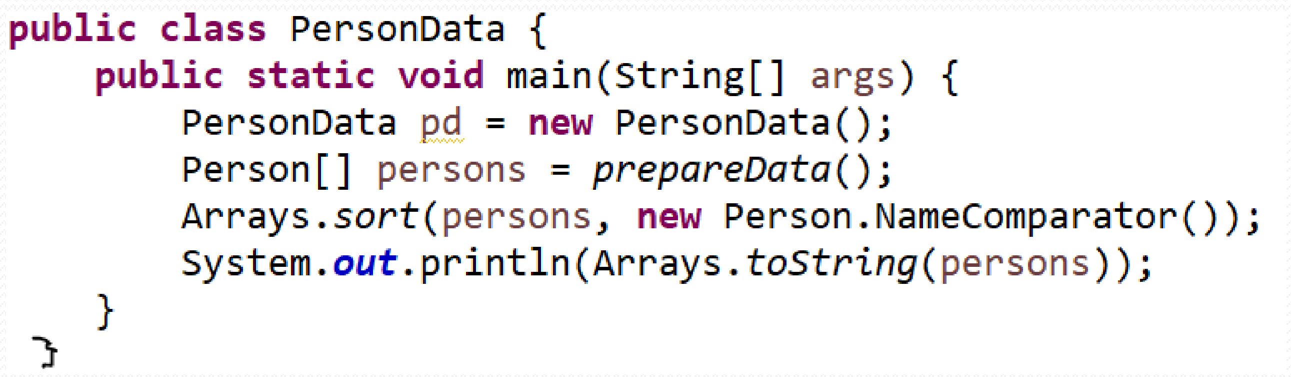
```
public class PersonData {
    private class NameComparator implements Comparator<Person>{
        @Override
        public int compare(Person p1, Person p2) {
            return p1.getName().compareTo(p2.getName());
        }
    }

    public static void main(String[] args) {
        PersonData pd = new PersonData();
        Person[] persons = prepareData();
        Arrays.sort(persons, pd.getNameComparator());
        System.out.println(Arrays.toString(persons));
    }
```

Implementing a Comparator as a Static Nested Class

- Since NameComparator does not need access to private Person data, NameComparator can be implemented as a static utility class inside Person



```java
public class Person {
    static class NameComparator implements Comparator<Person>{
        @Override
        public int compare(Person p1, Person p2) {
            return p1.getName().compareTo(p2.getName());
        }
    }
    private String name;
}
```



```java
public class PersonData {
    public static void main(String[] args) {
        PersonData pd = new PersonData();
        Person[] persons = prepareData();
        Arrays.sort(persons, new Person.NameComparator());
        System.out.println(Arrays.toString(persons));
    }
}
```

Outline of Topics

1. Definitions of Four Types of Nested and Inner Class
2. Most Commonly Used Nested Classes: Member and Static
3. Examples in Context of Sorting: The Comparator interface
4. **Local and Anonymous Inner Classes and Sorting**
5. Using Lambda Expressions in Place of Anonymous Inner Classes

Local Inner Classes

- Local inner classes are defined entirely within the body of a method.
- Access specifiers (public, private, etc) are not used to affect access of the inner class since access to the inner class is always restricted to the local access within the method body. (However, methods in the inner class may be – and may need to be – given some access specifier – see the example.)
- Local inner classes have access to instance variables and methods in the enclosing class; they also have access to *local variables* – variables inside the method body, as well as parameters passed in to the method – as long as these local variables are *effectively final* (this means that your code cannot change the values of these variables during execution).

  [Note: Prior to jdk 1.8, such variables had to be declared **final**. Examples are shown later in these slides.]

Sorting Using a Local Inner Class

- The NameComparator can be made even more private by embedding it into a sort method

```
public static void main(String[] args) {
    PersonData pd = new PersonData();
    Person[] persons = prepareData();
    pd.sort(persons);
    System.out.println(Arrays.toString(persons));
}

private void sort(Person[] persons) {
    class NameComparator implements Comparator<Person>{
        @Override
        public int compare(Person p1, Person p2) {
            return p1.getName().compareTo(p2.getName());
        }
    }
    Arrays.sort(persons, new NameComparator());
}
```

Exercise 5.2

Does the following code compile? If so, what happens when the main method is run? Is there a runtime error? If not, what is printed to the console?

```java
public class Outer {
    private int data = 10;
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.printVal(20);
    }
    void printVal(int bound) {
        if(data < bound) {
            class Inner {
                public int getValue() {
                    return data;
                }
            }
            Inner inner = new Inner();
            System.out.println("Inside inner: " + inner.getValue());
        } else {
            System.out.println("Inside outer: " + (data - bound));
        }
    }
}
```

Output:

```
Inside inner: 10
```

Advantages of Local Inner Classes

Two reasons for preferring local inner classes to member inner classes:

1. A local inner class is defined precisely where it is needed. This makes it easier to maintain, and makes code easier to follow

2. A local inner class can be used only for one purpose – the purpose of its enclosing method. Therefore, local inner classes are used to provide *strong encapsulation.*

   In the example, there is no reason to make the `NameComparator` accessible to any method (in any class) other than the method that actually does the sorting.

Issues Concerning Local Inner Classes

- *Auxiliary Method.* Sometimes the desired functionality of the inner class is not naturally associated with a method, so an additional method has to be created in order to specify a local inner class. This is what happened in the example – a `sort` method was introduced to permit the definition of a local inner class `NameComparator`.

- *Local Inner Classes Should Be Small.* When the inner class requires more than a small amount of code, it should not be squeezed inside the body of a method – the method body would become too big, harder to read, and maintenance of the method becomes more difficult. In such cases, member inner classes are preferable.

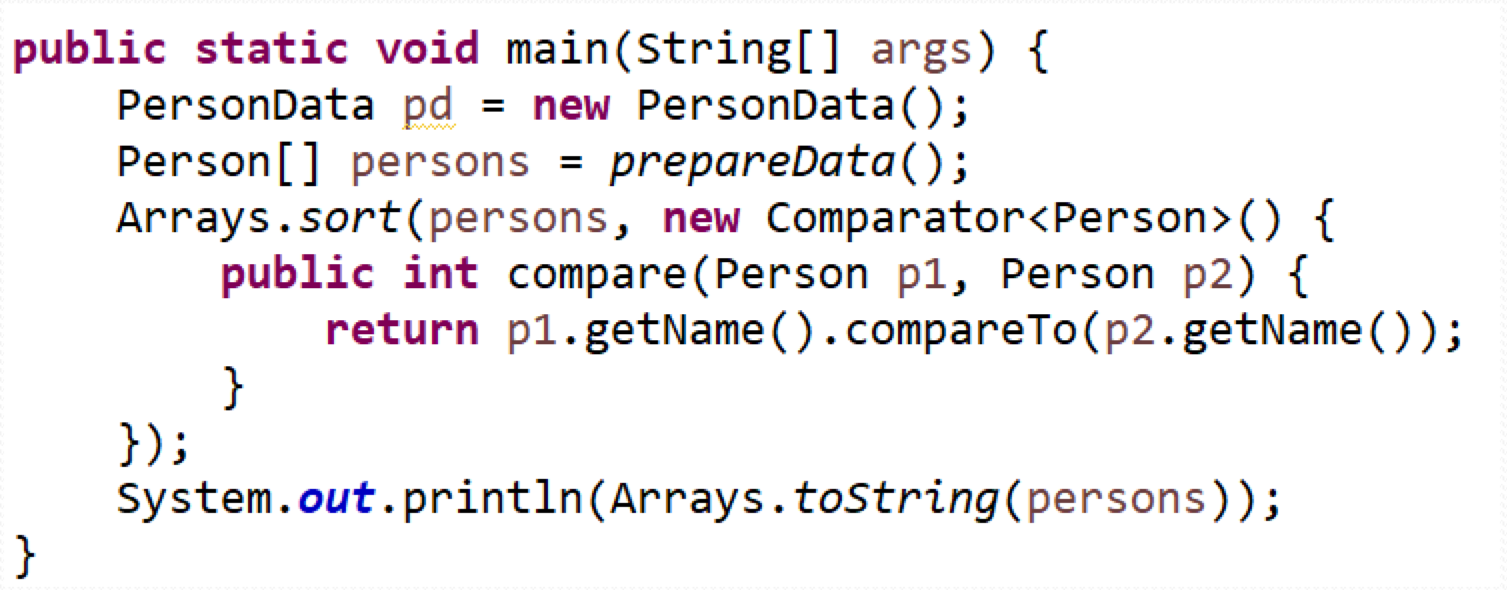
- *Avoid Including Local Inner Classes Involved in a Loop.* If the method in which the local inner class is defined is called from a loop, the inner class will be instantiated repeatedly. This behavior can be costly – in such cases, it is usually better to move the class out as a member inner class (or even a top-level class) and instantiate it just once; if its values need to take on different values as a loop executes, these can be set using setter methods.

Anonymous Inner Classes

- An anonymous inner class is a kind of inner class that is defined – without a name – and instantiated in a single block of code.
- Sometimes an anonymous inner class is defined – like local inner classes – within a method body. In that case, the code is even more compact, and has the same advantages as a local inner class.
- In general, an anonymous inner class is used to create an "on the fly" subclass of a known class or an "on the fly" implementation of a known interface.

Sorting Using an Anonymous Inner Class

- When the needed Comparator is defined "on the fly" within the Arrays.sort method, the code is very compact.



```java
public static void main(String[] args) {
    PersonData pd = new PersonData();
    Person[] persons = prepareData();
    Arrays.sort(persons, new Comparator<Person>() {
        public int compare(Person p1, Person p2) {
            return p1.getName().compareTo(p2.getName());
        }
    });
    System.out.println(Arrays.toString(persons));
}
```

Advantages to Anonymous Inner Classes

- They can be used in place of local inner classes without creating an artificial auxiliary method (as in the previous slide)
- They provide the same strong encapsulation as local inner classes.
- They have wider applicability than local inner classes: Whenever either a subclass of a class or an implementation of an interface is needed, anonymous inner classes can be used – no enclosing method is necessary.

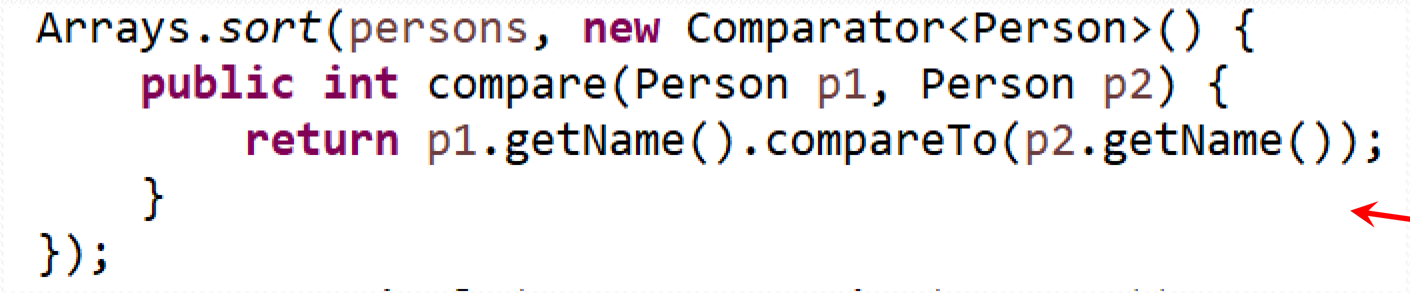
Disadvantages to Anonymous Inner Classes

- Explicit constructors cannot be used within an anonymous inner class (constructors give a name to a class and anonymous inner classes have no name)
- The syntax can be confusing – hard to read and maintain.

Outline of Topics

1. Definitions of Four Types of Nested and Inner Class
2. Most Commonly Used Nested Classes: Member and Static
3. Examples in Context of Sorting: The Comparator interface
4. Local and Anonymous Inner Classes and Sorting
5. **Using Lambda Expressions in Place of Anonymous Inner Classes**

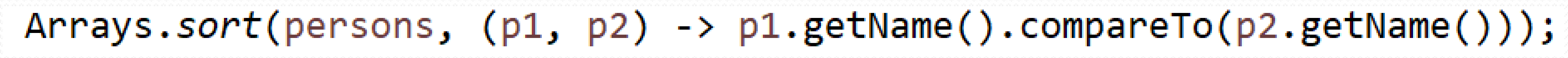
Example: Implementing a Comparator with a Lambda Expression (for jse8 and later)



```
Arrays.sort(persons, new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
});
```

using anonymous inner class

using a lambda expression



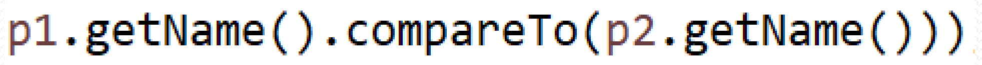
```
Arrays.sort(persons, (p1, p2) -> p1.getName().compareTo(p2.getName()));
```

About the Code Sample

- A lambda expression has been used to replace an implementation of the `Comparator` interface.

- <u>Note</u>: `Comparator<Person>` has just one (abstract) method `compare`, so it is a <u>functional interface</u>. The `compare` method takes two arguments of type `Person` and maps it to a block of code, representing the action to be performed.

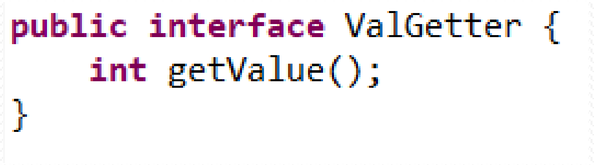
  The lambda expression just maps the pair `p1, p2` to a block of code



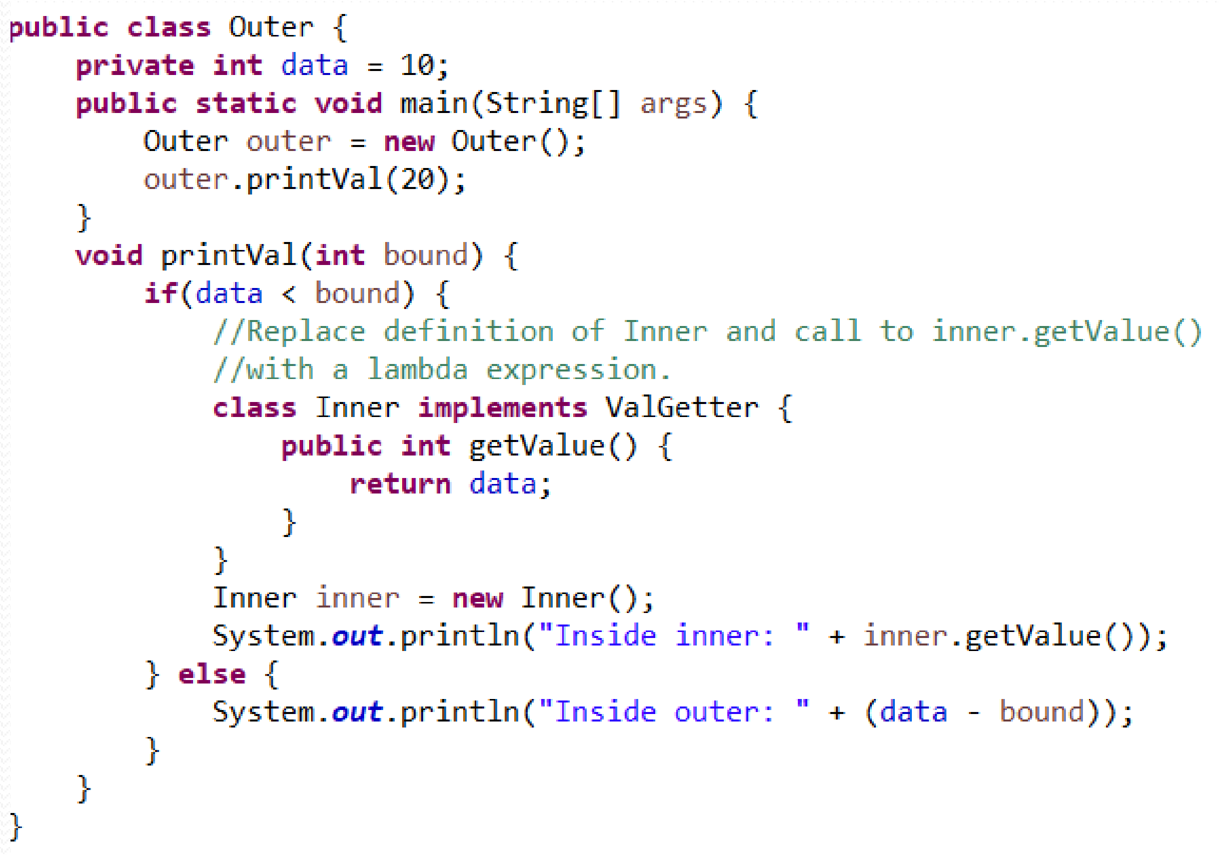
```
p1.getName().compareTo(p2.getName()))
```

Exercise 5.3

Refactor the code below so that the instance of the inner class is replaced by a *named* lambda expression. Notice that `ValGetter` is a functional interface. *Hint*: Look at how NameComparator was replaced by a lambda.



```
public interface ValGetter {
    int getValue();
}
```



```
public class Outer {
    private int data = 10;
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.printVal(20);
    }
    void printVal(int bound) {
        if(data < bound) {
            //Replace definition of Inner and call to inner.getValue()
            //with a lambda expression.
            class Inner implements ValGetter {
                public int getValue() {
                    return data;
                }
            }
            Inner inner = new Inner();
            System.out.println("Inside inner: " + inner.getValue());
        } else {
            System.out.println("Inside outer: " + (data - bound));
        }
    }
}
```

Exercise 5.3 - Solution

```
public class Outer {
    private int data = 10;
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.printVal(20);
    }
    void printVal(int bound) {
                if(data < bound) {
                            //Notice Inner is not accessible outside if block
            class Inner {
                public int getValue() {
                    return data;
                }
            }
            Inner inner = new Inner();
            System.out.println("Inside inner: " + inner.getValue());
        } else {
            System.out.println("Inside outer: " + (data - bound));
        }
    }
}
```

```
@FunctionalInterface
public interface ValGetter {
            int getValue();
}
```

Summary

Java has four kinds of nested classes: member, static, local and anonymous

- *Member inner classes* are used as private support within a class, much as instance variables and private methods are used. They have full access to the instance variables and methods of the enclosing class.
- *Static nested classes* are top-level classes that are naturally associated with their enclosing class, but have no special access to the data or behavior of the enclosing class.
- *Local inner classes* are defined entirely within a method body; *anonymous inner classes* make it possible to define a class at the moment that an instance of the class is created. Both types of inner classes are accessible only within the local context in which they are defined, resulting in an extreme form of encapsulation.
- When inner classes are used as implementers of *functional interfaces* – like `Comparator` – they can be replaced by *lambda expressions*, which extract from the inner class implementation the bare functional essence, resulting in more compact and easier to understand code.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Inner classes retain the memory of their "unbounded" context

1. A *nested class* is a class that is defined inside another class.
2. An *inner class* is a nested class that has full access to its context, its enclosing class.

---

3. **Transcendental Consciousness:** TC is the unbounded context for individual awareness.
4. **Wholeness moving within itself**: When individual awareness is permanently and fully established in its transcendental "context" – pure consciousness – every impulse of creation is seen to be an impulse of one's own awareness.