Victoria University of Wellington, New Zealand
Te Whare Wananga o te Upoko o te Ika a Maui
Aotearoa

SWEN221:
Software Development

17: Inner & Anonymous Classes

David J. Pearce & Nicholas Cameron & James Noble & Marco Servetto
Engineering and Computer Science, Victoria University

Static Inner classes
Hierarchical code organization

```
public interface Exp {







}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```
public interface Exp {
 public static class StringLiteral implements Exp{
  String value;
 }









}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```java
public interface Exp {
  public static class StringLiteral implements Exp{
    String value;
  }
  public static class FieldAccess implements Exp{
    Exp receiver; String fName;
  }







}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```
public interface Exp {
  public static class StringLiteral implements Exp{
    String value;
  }
  public static class FieldAccess implements Exp{
    Exp receiver; String fName;
  }
  public static class MethodCall implements Exp{
    Exp receiver; String mName; List<Exp> parameters;
  }




}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```
public interface Exp {
  public static class StringLiteral implements Exp{
    String value;
  }
  public static class FieldAccess implements Exp{
    Exp receiver; String fName;
  }
  public static class MethodCall implements Exp{
    Exp receiver; String mName; List<Exp> parameters;
  }
  public static class BinOp implements Exp{
    Op op; Exp left; Exp right;

  }

}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```
public interface Exp {
  public static class StringLiteral implements Exp{
    String value;
  }
  public static class FieldAccess implements Exp{
    Exp receiver; String fName;
  }
  public static class MethodCall implements Exp{
    Exp receiver; String mName; List<Exp> parameters;
  }
  public static class BinOp implements Exp{
    Op op; Exp left; Exp right;
    public static enum Op{ PLUS, MINUS, AND, OR, ... }
  }
  ...
}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```
class Foo {
  public static class Bar {...}
  private static class Beer {...}
  ...
}
public class Main {
  public static void main(String[ ] args){
    Foo.Bar bar= new Foo.Bar();
    System.out.println(bar);
    System.out.println(Exp.Op.PLUS);
  }
}
```

(Non-static) Inner classes, are much more complex, that Static Inner classes!

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

Static Inner classes
Hierarchical code organization

```
class Foo {
  public static class Bar {...}
  private static class Beer {...}
  ...
}
public class Main {
  public static void main(String[ ] args){
    Foo.Bar bar= new Foo.Bar();
    System.out.println(bar);
    System.out.println(Exp.Op.PLUS);
  }
}
```

Static Inner classes, often called nested classes in other languages, are just a way to do Hierarchical code organization.

Their type is simply a composed type, like `Exp.BinOp` or `Exp.BinOp.Op`

The fact that the static class `BinOp` is inside `Exp` have no operational semantic.

(Non-static) Inner classes, are much more complex, that Static Inner classes!

(Non-static) Inner Classes

- static inner classes → property of classes
  - a single `Foo.Bar` class exists
- inner classes → property of instances
  - each instance have its own (non-static) inner classes

- In the same way as

  static fields → property of classes

  fields → property of instances

Inner Classes: Example

```java
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;
  /* ... */
  public IntList() {this.data = new int[4];}
  public Iterator<Integer> iterator() {
    return new InternalIter();
  }
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;
    public boolean hasNext() {return pos < size;}
    public Integer next(){return data[pos++];}
    /* ... */
  }
}
```

Inner Class

Inner Classes: Example

```java
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;
  /* ... */
  public IntList() {this.data = new int[4];}
  public Iterator<Integer> iterator() {
    return new InternalIter();
  }
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;
    public boolean hasNext() {return pos < size;}
    public Integer next(){return data[pos++];}
    /* ... */
  }
}
```

Can access private fields/methods of enclosing class

Inner Classes: Example

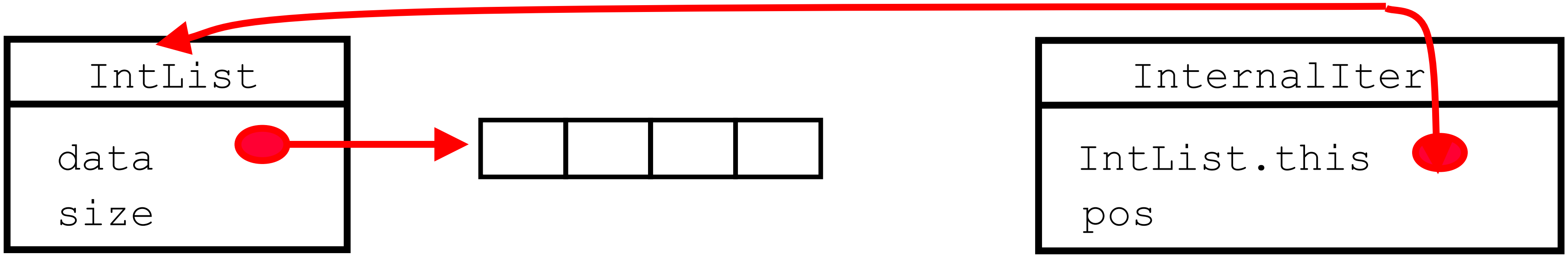
```
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;
  /* ... */
  public IntList() {this.data = new int[4];}
  public Iterator<Integer> iterator() {
    return new InternalIter();
  }
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;
    public boolean hasNext() {return pos < size;}
    public Integer next(){return data[pos++];}
    /* ... */
  }
}
```

Enclosing class can construct and return instances of inner class

Other classes cannot construct instances as it's private

Inner Classes: Scoping

- Inner classes have *outer pointer*
  - For accessing fields/methods of enclosing class (outer)
  - Outer pointer automatically supplied for new inner class



```
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;/* ... */
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;/* ... */
  }
}
```

Inner Classes: Explicit Scoping

```
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;
  /* ... */
  }
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;
    public Integer next(){
      return IntList.this.data[InternalIter.this.pos++];
    }
    /* ... */
  }
}
```

This line is now fully explicit in this-scoping

Inner Classes: Explicit Scoping

```
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;
  /* ... */
  }
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;
    public Integer next(){
      return this.data[IntList.this.pos++];
    }
    /* ... */
  }
}
```

Wrong explicit scoping here

Inner Classes: Explicit Scoping

```
class IntList implements Iterable<Integer> {
  private int[] data;
  private int size = 0;
  /* ... */
  }
  private class InternalIter implements Iterator<Integer>{
    private int pos = 0;
    public Integer next(){
      return InternalIter.this.data[this.pos++];
    }
    /* ... */
  }
}
```

Wrong explicit scoping here

Inner Classes: External Construction

```
class Shape {
  /* ... */
  public class Square {
    private int x, y, width, height;
    public Square(int x, int y, int width, int height){
      /* ... */
    }
  }
}
    /* ... */
    Shape outer = new Shape();
    Shape.Square square = outer.new Square(0,0,8,42);
    square = new Shape().new Square(0,0,8,42);
```

- External Construction
  - If constructing inner class outside outer, or in static method, must supply outer pointer **explicitly**

Inner Classes - Static Inner Classes

- Static Inner Classes have no outer pointer!
  - So, can not access fields/methods of enclosing class
  - But, can construct without providing outer pointer
  - If no need to access enclosing info, then this is more convenient (and potentially more efficient)

Inner Classes - Static Inner Classes

- Static Inner Classes have no outer pointer!
  - So, can not access fields/methods of enclosing class
  - But, can construct without providing outer pointer
  - If no need to access enclosing info, then this is more convenient (and potentially more efficient)
- (Non-Static) Inner Classes have outer pointer!
  - So, they have multiple `this` and can use it to (implicitly/explicitly) access fields/methods of enclosing instance
  - But, can not be instantiated without providing the outer pointer

Method Local Inner Classes

```java
class Outer {
  public Outer create(final int field) {
   class Inner extends Outer {
       private int myfield = field;
       /*...*/
      };

   return new Inner();
  }
 }
```

Non-static method local class

Can access local variables + parameters provided they are (effectively) final.

- Can even define classes within a method!
  - These are only visible within that method
  - But, their instances can still be returned
  - Cannot have static method-local classes

Anonymous Classes: Example

```
public static void main(String[] args) {
  List<String> myList = new ArrayList<String>(){
      // override ArrayList.add
      public boolean add(String x) {
        System.out.println("ADDED: " + x);
        return super.add(x);
       }
      }
     };
 }
```

- Anonymous Class
  - Has no class definition and, hence, no name
  - Defined as an extension of existing class
  - Can override methods and/or define fields

Anonymous Class

Anonymous Classes: Syntax

```
public class Test {
   private int field;
   public Test(int field) { this.field = field; }
   public void aMethod() { /*...*/ }

   public static void main(String[] args) {
    Test x = new Test() {
        public void aMethod() {
         System.out.println("GOT HERE");
        }
       };
   }
 }
```

Compile time error

Anonymous Classes: Syntax

```
public class Test {
   private int field;
   public Test(int field) { this.field = field; }
   public void aMethod() { /*...*/ }

   public static void main(String[] args) {
    Test x = new Test(1) {
        public void aMethod() {
         System.out.println("GOT HERE");
        }
       };
   }
  }
```

Can provide arguments to super constructor

Anonymous Classes: Syntax

```
public class Test {
   private int field;
   public Test(int field) { this.field = field; }
   public void aMethod() { /*...*/ }

   public static void main(final String[] a) {
    Test x = new Test(1) {
        public void aMethod() {
         System.out.println("GOT "+a+" "+field);
        }
       };
   }
  }
```

Can access local variables and parameters
(provided they are effectively final)
and enclosing fields

Anonymous Classes: Interfaces

```
ArrayList<String> ls=/*...*/;
Collections.sort(ls,new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
    }});
```

Can even make anonymous class from an interface!!!

Very common case: interface with just 1 method.
Lambdas are just a syntactic sugar for anonymous classes

Anonymous Classes: Interfaces

```
ArrayList<String> ls=/*...*/;
Collections.sort(ls,new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
    }});
```

- Learn how to read the code through the syntax: fading away the anonymous class+method declaration, what you obtain read like:

  Sort `ls` using `s1.compareToIgnoreCase(s2)`

Anonymous Classes: Interfaces

```
ArrayList<String> ls=/*...*/;
Collections.sort(ls,new Comparator<String>(){
  public int compare(String s1, String s2) {
    return s1.compareToIgnoreCase(s2);
    }});
```

- Learn how to read the code through the syntax: fading in the opposite way, we see
  - type informations-- they double check

    that we know what we are doing

```
int Comparator<String>.compare(String,String)
```

  - names: s1,s2 -- can be used to identify concepts

Before Java8: Used a lot for event handlers

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;

public class MiniGui extends JFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        MiniGui g = new MiniGui();
        g.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        g.getRootPane().setLayout(new BorderLayout());
        JButton b = new JButton("------Bar-------");
        b.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            System.out.println("Button pressed");
          }});
        g.getRootPane().add(b, BorderLayout.CENTER);
        g.pack();
        g.setVisible(true);
      }});
  }}
```

Before Java8: Used a lot for event handlers

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;

public class MiniGui extends JFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        MiniGui g = new MiniGui();
        g.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        g.getRootPane().setLayout(new BorderLayout());
        JButton b = new JButton("------Bar------");
        b.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e) {
            System.out.println("Button pressed");
          }});
        g.getRootPane().add(b, BorderLayout.CENTER);
        g.pack();
        g.setVisible(true);
      }});
  }}
```

After Java8: Lambdas for event handlers

```
import java.awt.*; import java.awt.event.*; import javax.swing.*;

public class MiniGui extends JFrame {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        MiniGui g = new MiniGui();
        g.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        g.getRootPane().setLayout(new BorderLayout());
        JButton b = new JButton("------Bar-------");
        b.addActionListener(

        e->{System.out.println("Button pressed");}
            );
        g.getRootPane().add(b, BorderLayout.CENTER);
        g.pack();
        g.setVisible(true);
      }});
  }}
```

Example: Groups of Persons

-A Person have a String name and many Person friends.

-A Group of friend represent a set of friendships.

We want to enforce the following:

-A Person only belong to a single Group, and have friends only in that Group.

```java
public class Group{//facade pattern
  private List<Person> ps=new ArrayList<>();
  public List<Person> getPersons(){
    return Collections.unmodifiableList(ps);}
  public void addPerson(String name){ps.add(new Person(name));}
  private void checkInGroup(Person p) {
    if(!ps.contains(p)) {throw new Error("");}
    }
  public void connect(Person p1,Person p2) {
    checkInGroup(p1);  checkInGroup(p2);
    if(p1==p2) {return;}
    if(p1.friends.contains(p2)) {return;}
    p1.friends.add(p2);  p2.friends.add(p1);
    }
  public final static class Person{//important: nested
    private String name;
    public String getName() {return this.name;}
    private Person(String name) {this.name=name;}
    private List<Person> friends=new ArrayList<>();
    public List<Person> getFriends(){
      return Collections.unmodifiableList(friends);}
    public String toString() {return ...;}
    } }
```

```
public class Group{//facade pattern
  private List<Person> ps=new ArrayList<>();
  public List<Person> getPersons(){
    return Collections.unmodifiableList(ps);}
  public void addPerson(String name){ps.add(new Person(name));}
  private void checkInGroup(Person p) {
    if(!ps.contains(p)) {throw new Error("");}
    }
  public void connect(Person p1,Person p2) {
    checkInGroup(p1);  checkInGroup(p2);
    if(p1==p2) {return;}
    if(p1.friends.contains(p2)) {return;}
    p1.friends.add(p2);  p2.friends.add(p1);
    }
  public final static class Person{//important: nested
    private String name;
    public String getName() {return this.name;}
    private Person(String name) {this.name=name;}
    private List<Person> friends=new ArrayList<>();
    public List<Person> getFriends(){
      return Collections.unmodifiableList(friends);}
    public String toString() {return ...;}
    } }
```

```java
public class Group{//facade pattern
  private List<Person> ps=new ArrayList<>();
  public List<Person> getPersons(){
    return Collections.unmodifiableList(ps);}
  public void addPerson(String name){ps.add(new Person(name));}
  private void checkInGroup(Person p) {
    if(!ps.contains(p)) {throw new Error("");}
    }
  public void connect(Person p1,Person p2) {
    checkInGroup(p1);  checkInGroup(p2);
    if(p1==p2) {return;}
    if(p1.friends.contains(p2)) {return;}
    p1.friends.add(p2);  p2.friends.add(p1);
    }
  public final static class Person{//important: nested
    private String name;
    public String getName() {return this.name;}
    private Person(String name) {this.name=name;}
    private List<Person> friends=new ArrayList<>();
    public List<Person> getFriends(){
      return Collections.unmodifiableList(friends);}
    public String toString() {return ...;}
    } }
```

```java
public class Group{//facade pattern
  private List<Person> ps=new ArrayList<>();
  public List<Person> getPersons(){
    return Collections.unmodifiableList(ps);}
  public void addPerson(String name){ps.add(new Person(name));}
  private void checkInGroup(Person p) {
    if(!ps.contains(p)) {throw new Error("");}
    }
  public void connect(Person p1,Person p2) {
    checkInGroup(p1);  checkInGroup(p2);
    if(p1==p2) {return;}
    if(p1.friends.contains(p2)) {return;}
    p1.friends.add(p2);  p2.friends.add(p1);
    }
  public final static class Person{//important: nested
    private String name;
    public String getName() {return this.name;}
    private Person(String name) {this.name=name;}
    private List<Person> friends=new ArrayList<>();
    public List<Person> getFriends(){
      return Collections.unmodifiableList(friends);}
    public String toString() {return ...;}
    } }
```

```
public class Group{//facade pattern
  private List<Person> ps=new ArrayList<>();
  public List<Person> getPersons(){
    return Collections.unmodifiableList(ps);}
  public void addPerson(String name){ps.add(new Person(name));}
  private void checkInGroup(Person p) {
    if(!ps.contains(p)) {throw new Error("");}
    }
  public void connect(Person p1,Person p2) {
    checkInGroup(p1);  checkInGroup(p2);
    if(p1==p2) {return;}
    if(p1.friends.contains(p2)) {return;}
    p1.friends.add(p2);  p2.friends.add(p1);
    }
  public final static class Person{//important: nested
    private String name;
    public String getName() {return this.name;}
    private Person(String name) {this.name=name;}
    private List<Person> friends=new ArrayList<>();
    public List<Person> getFriends(){
      return Collections.unmodifiableList(friends);}
    public String toString() {return ...;}
    } }
```

Private means: private to the whole set of nested classes

Groups of Persons

How is up to now?

- Success?
  - Can the user modify Person directly?
    - no, all mutation operation are private, so **only Groups can modify them**.
  - **code logic** check that persons of different groups are never connected.
  - Persons are **created by the Group**, and not directly by the user.
- A delicate balance! Change one thing, all collapse!
- Of course, we are assuming to run under a **SecurityMangager** preventing nasty reflection tricks
  (not like the default one)

A more usable Group

- How to expose only a read view of a group:

ReadGroup is an interface, Group will implement it, and there will be a method to wrap a Group in a ReadGroup.

Note how there is **no way** to extract the inner group from the result of a ReadGroup.of(..)

```java
public class Group implements ReadGroup{...}

public interface ReadGroup {
  public List<Group.Person> getPersons();
  public static ReadGroup of(Group g) {
    return new ReadGroup(){
      public List<Group.Person> getPersons(){
        return g.getPersons();
        }
      };
    }
  }
```

A more usable Group

- How offer a deepClone() operation:
  Hard to do by hand: circular graph of friends!
  Java idiomatic way: use serialization!

```
public class Group implements ReadGroup,Serializable{...
  public final static class Person implements Serializable{..}
  public Group deepClone() {return DeepClone.copy(this);} }
class DeepClone{
  @SuppressWarnings("unchecked")public static <T> T copy(T orig){
    ByteArrayOutputStream aux=new ByteArrayOutputStream();
    try(ObjectOutputStream out= new ObjectOutputStream(aux)){
      out.writeObject(orig);
      out.flush(); }
    catch(IOException e) {throw new Error(e);}
    try(ObjectInputStream in = new ObjectInputStream(
        new ByteArrayInputStream(aux.toByteArray()))){
      return (T)in.readObject(); }
    catch(IOException|ClassNotFoundException e) {
      throw new Error(e); }
    }  }
```

Group: testing/example usage

```
String s(ReadGroup g) {return g.getPersons().toString();}

@Test public void test() {
  Group g=new Group();
  g.addPerson("Bob");//persons created by the group
  g.addPerson("Alice");
  //modification handled by the group
  g.connect(g.getPersons().get(0), g.getPersons().get(1));
  assertEquals("[Bob[Alice], Alice[Bob]]",s(g));

  ReadGroup wrapper=ReadGroup.of(g);//read only view
  ReadGroup imm=ReadGroup.of(g.deepClone());//immutable datatype
  assertEquals("[Bob[Alice], Alice[Bob]]",s(wrapper));
  assertEquals("[Bob[Alice], Alice[Bob]]",s(imm));

  g.addPerson("Wally");//modifies only wrapper
  assertEquals("[Bob[Alice], Alice[Bob], Wally[]]",s(wrapper));
  assertEquals("[Bob[Alice], Alice[Bob]]",s(imm));
  }
```

quiz

```
public class Exercise {
 static int x=1; int y=2;
 static int z=0;
 static class Foo{
  static int y=3; int x=4;
  static int m1(){
   return Foo.y+Exercise.x;}
  int m2(){
   return y+x;}
  class Bar{
   int x=5; int y=6;
   int m3(){
    return y+x+m1()+m2();}
   int m4(){
    return z+Foo.this.y
    +Foo.this.x;}
  }}
```

```
public static void main(String[] args){
 Foo foo=new Foo();
 Foo.Bar bar=foo.new Bar();
 System.out.println(foo.m1());
 System.out.println(foo.m2());
 System.out.println(bar.m3());
 System.out.println(bar.m4());
 }}
```