Nested Classes

- some classes exist only to help out with the implementation of another class
  - e.g. `DoubleListNode` is used only within `SolitaireDeck`
- a *nested class* is a class defined inside another class
  - *static nested class*
  - *inner class* – a non-static nested class
  - *anonymous inner class* – unnamed inner class
  - *local class* – class defined within a method
- technically nested classes can be `public`
  - typically most appropriate for a nested interface or abstract class rather than a concrete class in order to maintain encapsulation and information hiding

Syntax – Declaration

- static nested class

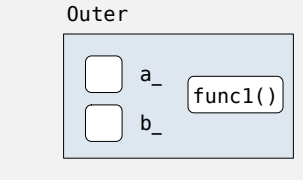
```
public class Outer {
  static private class Nested {
    …
  }
  …
}
```

- inner class

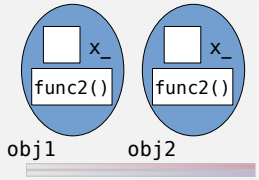
```
public class Outer {
  private class Inner {
    …
  }
  …
}
```


Semantics

- `static` means there is only one copy shared by all instances of that type
- non-`static` means there is a separate copy per object

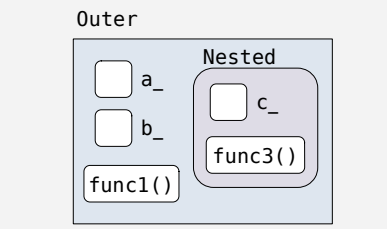



```
public class Outer {
  static int a_, b_;
  int x_;

  static void func1 () { … }
  void func2 () { … }
}
```

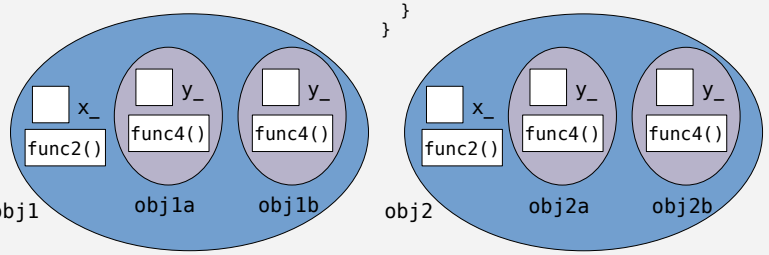


static variables `a_`, `b_` are shared by `obj1` and `obj2`
static method `func1()` can only access `a_`, `b_`

`obj1`, `obj2` have their own copies of non-static variable `x_`
`obj1`, `obj2` have their own copies of non-static method `func2()` which can access `a_`, `b_`, and their own copy of `x_`

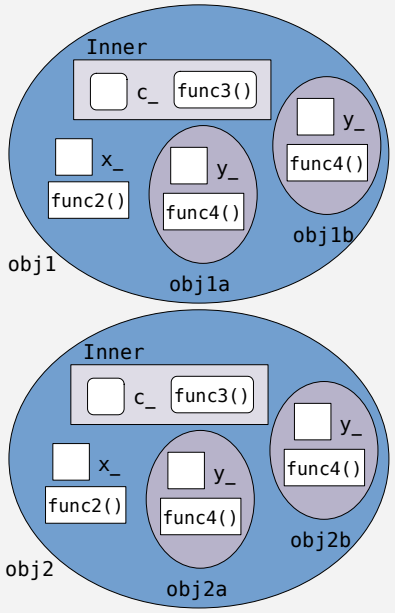


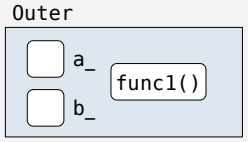

```
public class Outer {
  static int a_, b_;
  int x_;

  static void func1 () { … }
  void func2 () { … }

  static class Nested {
    static int c_;
    int y_;

    static void func3 () { … }
    void func4 () { … }
  }
}
```



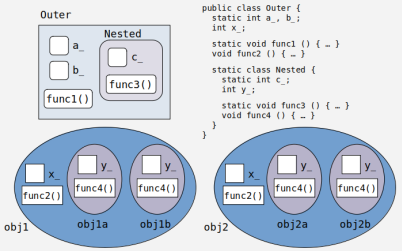
`obj1a`, `obj1b`, `obj2a`, `obj2b` share static variable `c_` and static method `func3()`
static method `func3()` can only access `a_`, `b_`, `c_`
`obj1a`, `obj1b`, `obj2a`, `obj2b` have their own copies of non-static variable `y_` and non-static method `func4()` which can access `a_`, `b_`, `c_`, and their own copies of `y_`

```
public class Outer {
  static int a_, b_;
  int x_;

  static void func1 () { … }
  void func2 () { … }

  class Inner {
    static int c_;
    int y_;

    static void func3 () { … }
    void func4 () { … }
  }
}
```



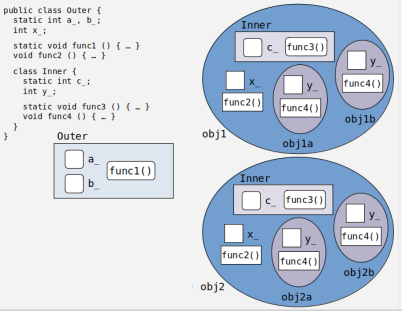


obj1a, obj1b, obj2a, obj2b share `static` variable `c_` and `static` method `func3()`
`static` method `func3()` can only access `a_`, `b_`, `c_`
obj1a, obj1b, obj2a, obj2b have their own copies of non-`static` variable `y_` and non-`static` method `func4()` which can access `a_`, `b_`, `c_`, obj1's or obj2's copy of `x_`, and their own copies of `y_`



`static` means there is only one copy shared by all instances of that type

- can be used outside the containing class with the syntax `Outer.Nested`
- can be used inside any methods of the containing class to create objects
- can access only `static` members of the containing class
- containing class can access only `static` members of the contained class



non-`static` means there is a separate copy per object

- can be used outside the containing class with the syntax `obj.Inner`
- can only be used inside non-`static` methods of the containing class to create objects
- can access any members of the containing class
- containing class can only access `static` members of the contained class

Semantics

- nested classes have access to instance variables and methods in the containing class
  - even `private` ones!
- the containing class has access to instance variables and methods in the nested class
  - even `private` ones!

can be used inside static methods of the containing class to create objects	static nested
can be used inside non-static methods of the containing class to create objects	both
can be used outside the containing class with the syntax ContainingClass.InsideClass	static nested
can be used outside the containing class with the syntax obj.InsideClass	inner
can be public	both
can be private	both
can access static members of the containing class	both
can access non-static members of the containing class	inner
can access private members of the containing class	both
containing class can access its static members	both
containing class can access its non-static members	neither
containing class can access its private members	both

Guidelines

- `static` nested class or inner class?
  - prefer a `static` nested class unless it needs to access non-`static` members of the containing class
- just because you have can do something doesn't mean you need to
  - treat nested classes like other classes – unless the class is purely a collection of variables with only getters/setters, avoid direct access to the nested class instance variables
    - provide getters, setters, and other methods as needed
  - limit public exposure of nested classes – don't break encapsulation!
- limit nested classes to short helper classes
  - you *could* make every class a nested class in the main program, but that makes the main class very long and prevents reuse of any of the classes
  - for longer classes that are specialized to the project rather than one class, use packages and package access

Examples

- purely internal helper class
  - `DoubleListNode` in `SolitaireDeck`
- helper class with some public exposure
  - `TreeNode` in `BinaryTree`


```
public class SolitaireDeck {

  private DoubleListNode deck_;
  private int size_;

  private class DoubleListNode {

    private SolitaireCard card_; // the card
    private DoubleListNode next_, prev_; // next and previous nodes

    private DoubleListNode ( SolitaireCard card ) {
      card_ = card;
      next_ = null;
      prev_ = null;
    }

    private DoubleListNode ( SolitaireCard card, DoubleListNode prev,
                             DoubleListNode next ) {
      card_ = card;
      prev_ = prev;
      next_ = next;
    }
  }

  public SolitaireDeck ( int size ) {
    size_ = size;
    deck_ = new DoubleListNode(new SolitaireCard(1));
  }

}
```

`DoubleListNode` is just a collection of variables, so it is possible to dispense with the getters and setters

(keep the constructors for convenience)


```
/**
 * A proper binary tree (i.e. every internal node has exactly two children).
 */
public class BinaryTree {

  private TreeNode root_;
  private int size_;

  /**
   * Create a new binary tree with one node.
   */
  public BinaryTree () {
    root_ = new TreeNode(this);
    size_ = 1;
  }

  /**
   * Create a new binary tree with one node sto
   *
   * @param element
   *          the element to be stored in the r
   */
  public BinaryTree ( int element ) {
    root_ = new TreeNode(element,this);
    size_ = 1;
  }

  /**
   * Get the root of the tree.
   *
   * @return the root of the tree
   */
  public Node getRoot () {
    return root_;
  }
```

```
  /**
   * Get the parent of a node.
   *
   * @param node
   *          the node (the node must belong to this tree, and not be null or
   *          the root)
   * @return the parent of the node
   */
  public Node getParent ( Node node ) {
    TreeNode treenode = checkNode(node);
    if ( node == root_ ) {
      throw new IllegalArgumentException("cannot get the parent "
          + "of the root");
    }

    return treenode.parent_;
  }

  /**
   * Get the left child of a node.
   *
   * @param node
   *          the node (the node must belong to this tree, and not be null or a
   *          leaf)
   * @return the left child of the node
   */
  public Node getLeftChild ( Node node ) {
    TreeNode treenode = checkNode(node);
    if ( isLeaf(node) ) {
      throw new IllegalArgumentException("cannot get the child " + "of a leaf");
    }

    return treenode.left_;
  }
```

```java
/**
 * An abstraction of the idea of a node in a tree, allowing for different
 * implementations of the tree (e.g. linked structure or array). Outside the
 * tree, the only thing one can do with a node is access its element.
 */
public interface Node {

  /**
   * Retrieve the element stored in this position.
   *
   * @return element stored in this position
   */

  public int getElement ();
}
```

we need a `public` Node type because `BinaryTree` operations need to take and return nodes, but the tree structure itself should be encapsulated within `BinaryTree`

the solution is to make `Node` a `public` interface with limited operations – just `getElement()` – and a `private` `TreeNode` class that implements that interface

– a `TreeNode` object is returned from `BinaryTree` but since the caller can only see it as the declared type – `Node` – they can't access its internals

both `Node` and `TreeNode` are inner classes in `BinaryTree`

```java
  /**
   * A node of the tree.
   */
  private class TreeNode implements Node {

    private int element_;

    private TreeNode parent_;
    private TreeNode left_, right_;

    private BinaryTree tree_;

    private TreeNode ( BinaryTree tree ) {
      tree_ = tree;
      parent_ = null;
      left_ = null;
      right_ = null;
    }

    private TreeNode ( int element, BinaryTree tree ) {
      element_ = element;
      tree_ = tree;
      parent_ = null;
      left_ = null;
      right_ = null;
    }

    @Override
    public int getElement () {
      return element_;
    }

  }
}
```

```java
public class BTDemo {

  public static void main ( String[] args ) {

    // create a tree with 20 at the root
    BinaryTree tree = new BinaryTree(20);

    // add 10 and 5 as the children of 20
    BinaryTree.Node root = tree.getRoot(); // the node with 20
    tree.expandLeaf(root,10,5);

    BinaryTree.Node left = tree.getLeftChild(root); // the node with 10

    // add 16 and 8 as the children of 10
    tree.expandLeaf(left,16,8);

    // add dummy nodes (no elements) as the children of 5 and 16
    tree.expandLeaf(tree.getRightChild(root));
    tree.expandLeaf(tree.getLeftChild(left));

    // add 7 as the left child of 8 (and a dummy node as the right child)
    BinaryTree.Node leftright = tree.getRightChild(left); // the node with 8
    tree.expandLeaf(leftright);
    tree.setElement(tree.getLeftChild(leftright),7);

    // add dummy nodes (no elements) as the
    tree.expandLeaf(tree.getLeftChild(leftr
```

using `BinaryTree`

```java
  /**
   * Return the leftmost internal node in the tree.
   *
   * @param tree
   *          the tree (size > 1)
   * @return the leftmost internal node
   */
  public static BinaryTree.Node findLeftmost ( BinaryTree tree ) {
    if ( tree.getSize() <= 1 ) {
      throw new IllegalArgumentException("tree must have more than one node; size "
          + tree.getSize());
    }

    // pattern: moving down the tree, interested in only one child
    BinaryTree.Node current = tree.getRoot();
    for ( ; !tree.isLeaf(tree.getLeftChild(current)) ; ) {
      current = tree.getLeftChild(current);
    }

    return current;
  }
```