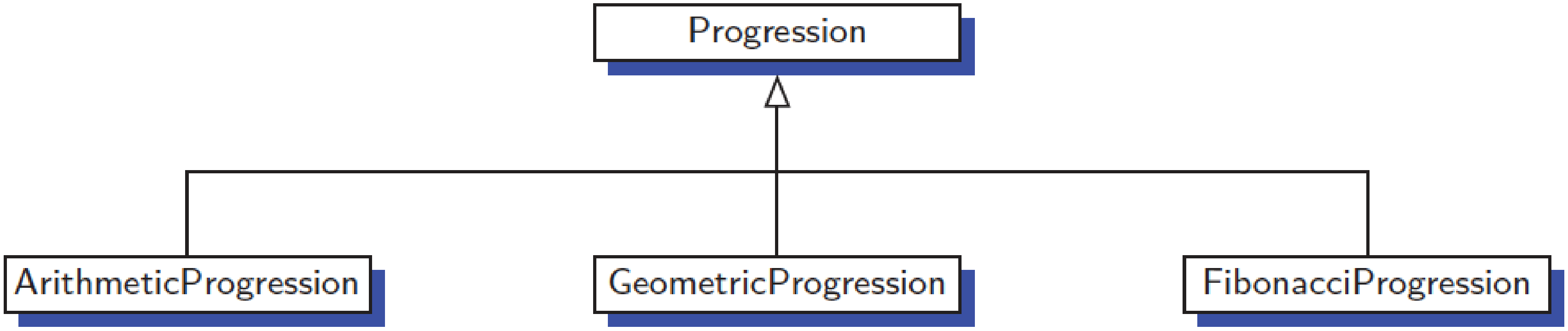
# Data Structures and Algorithms

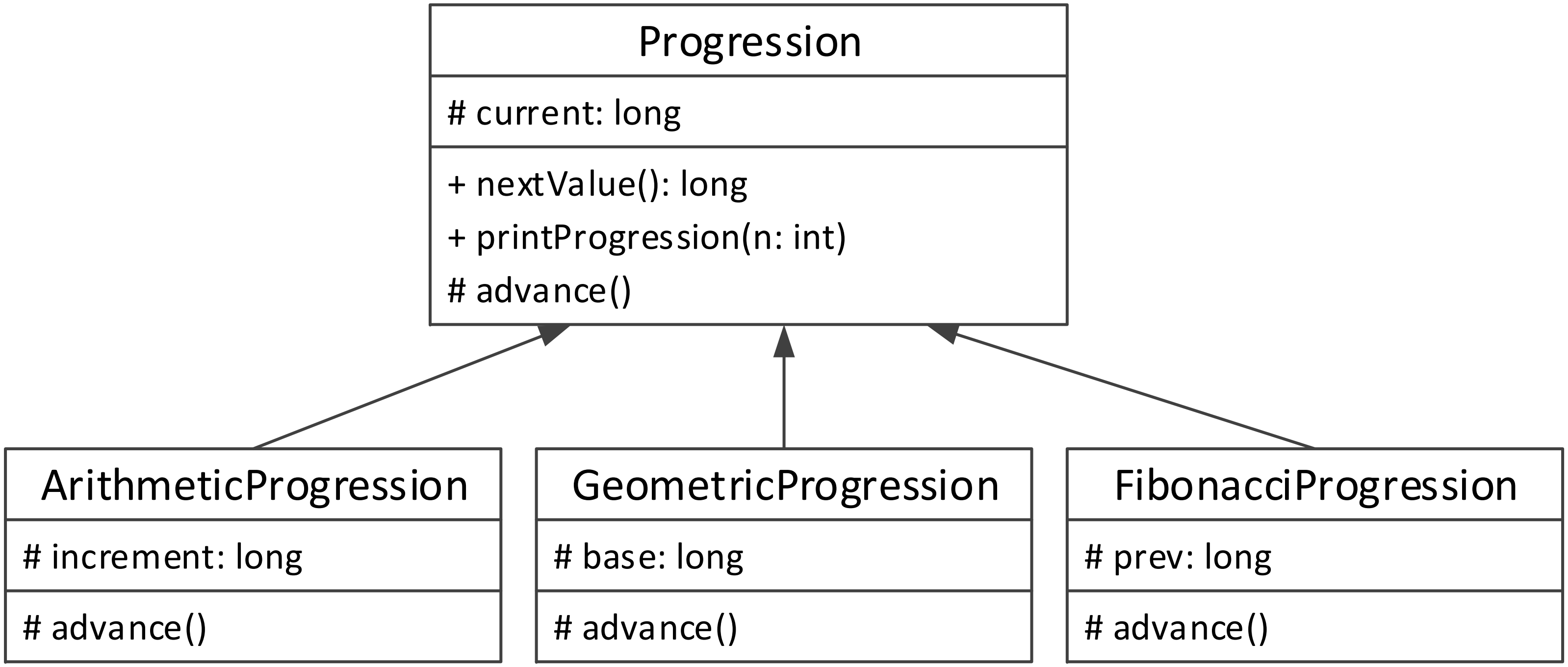
Chapter 2

# Inheritance

- Inheritance hierarchy example



# Inheritance



Progression

# current: long

+ nextValue(): long

+ printProgression(n: int)

# advance()

ArithmeticProgression

# increment: long

# advance()

GeometricProgression

# base: long

# advance()

FibonacciProgression

# prev: long

# advance()

# Inheritance

```
/* not a complete code */
public class Progression {

    protected long current;
    public Progression( ) { this(0); }
    public Progression(long start) { current = start; }
    public long nextValue( ) {
    }
    protected void advance( ) {
    }
    public void printProgression(int n) {
    }
}
```

# Inheritance

```
/* not a complete code */
public class ArithmeticProgression extends Progression {

    protected long increment;
    public ArithmeticProgression( ) { this(1, 0); }
    public ArithmeticProgression(long stepsize) {
    }
    public ArithmeticProgression(long stepsize, long start) {
    }
    protected void advance( ) {
    }
}
```

# Inheritance

```
/* not a complete code */
public class FibonacciProgression extends Progression {

    protected long prev;
    public FibonacciProgression( ) { this(0, 1); }
    public FibonacciProgression(long first, long second) {
    }
    protected void advance( ) {
    }
}
```

# Interface

- Used to specify a “contract” between different programs.
- No data
- Methods do not have implementation.
- Cannot be instantiated.
- Can be used for multiple inheritance.
- A class implementing an interface must implements all methods.

# Interface

```
/** Interface for objects that can be sold. */
public interface Sellable {
  /** Returns a description of the object. */
  public String description( );
  /** Returns the list price in cents. */
  public int listPrice( );
  /** Returns the lowest price in cents we will accept. */
  public int lowestPrice( );
}
```

# Interface

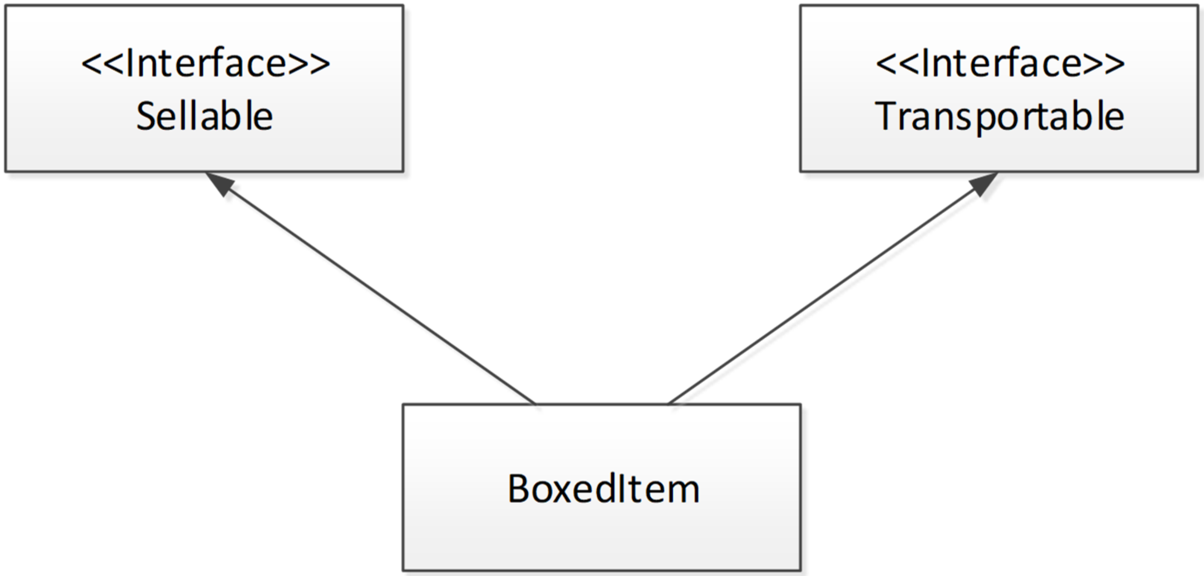
```
/** Class for photographs that can be sold. */
public class Photograph implements Sellable {
  private String descript;                        // description of this photo
  private int price;                              // the price we are setting
  private boolean color;                          // true if photo is in color
  public Photograph(String desc, int p, boolean c) {  // constructor
    descript = desc;
    price = p;
    color = c;
  }
  public String description( ) { return descript; }
  public int listPrice( ) { return price; }
  public int lowestPrice( ) { return price/2; }
  public boolean isColor( ) { return color; }
}
```

# Interface

```
/** Interface for objects that can be transported. */
public interface Transportable {
  /** Returns the weight in grams. */
  public int weight();
  /** Returns whether the object is hazardous. */
  public boolean isHazardous();
}
```

# Interface

- Multiple inheritance



# Interface

```
public class BoxedItem implements Sellable, Transportable {
  private String descript;     // description of this item
  private int price;           // list price in cents
  private int weight;          // weight in grams
  private boolean haz;         // true if object is hazardous
  private int height=0;        // box height in centimeters
  private int width=0;         // box width in centimeters
  private int depth=0;         // box depth in centimeters
  public BoxedItem(String desc, int p, int w, boolean h) {
    descript = desc;
    price = p;
    weight = w;
    haz = h;
  }
/* continue to the next slide */
```

# Interface

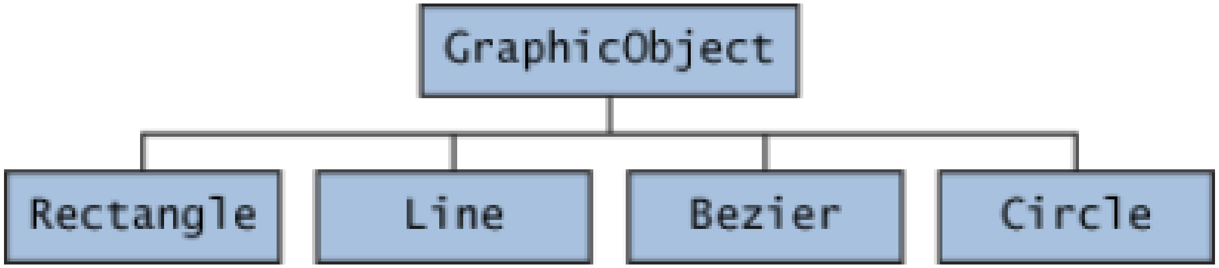
```
  public String description() { return descript; }
  public int listPrice() { return price; }
  public int lowestPrice() { return price/2;  }
  public int weight() { return weight; }
  public boolean isHazardous() { return haz; }
  public int insuredValue() { return price*2; }
  public void setBox(int h, int w, int d) {
    height = h;
    width = w;
    depth = d;
  }
}
```

# Abstract Class

- An abstract method: a method without implementation.
- A concrete method: a method with implementation.
- Abstract class:
  - Declared with *abstract* keyword.
  - May or may not have abstract method.
  - A class with an abstract method must be an abstract class.
  - Used when subclasses share many common variables and methods.
  - Cannot be instantiated.

# Abstract Class

- An example from Oracle documentation (https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html)



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

# Abstract Class

```
abstract class GraphicObject {
    int x, y;
    . . .
    void moveTo(int newX, int newt) {
        . . .
    }
    abstract void draw();
    abstract void resize();
}
```

# Abstract Class

```
class Rectangle extends GraphicObject {
    void draw() {
        // implementation
        . . .
    }
    void resize() {
        // implementation
        . . .
    }
}
```

# Interface and Abstract Class

- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.

# Interface and Abstract Class

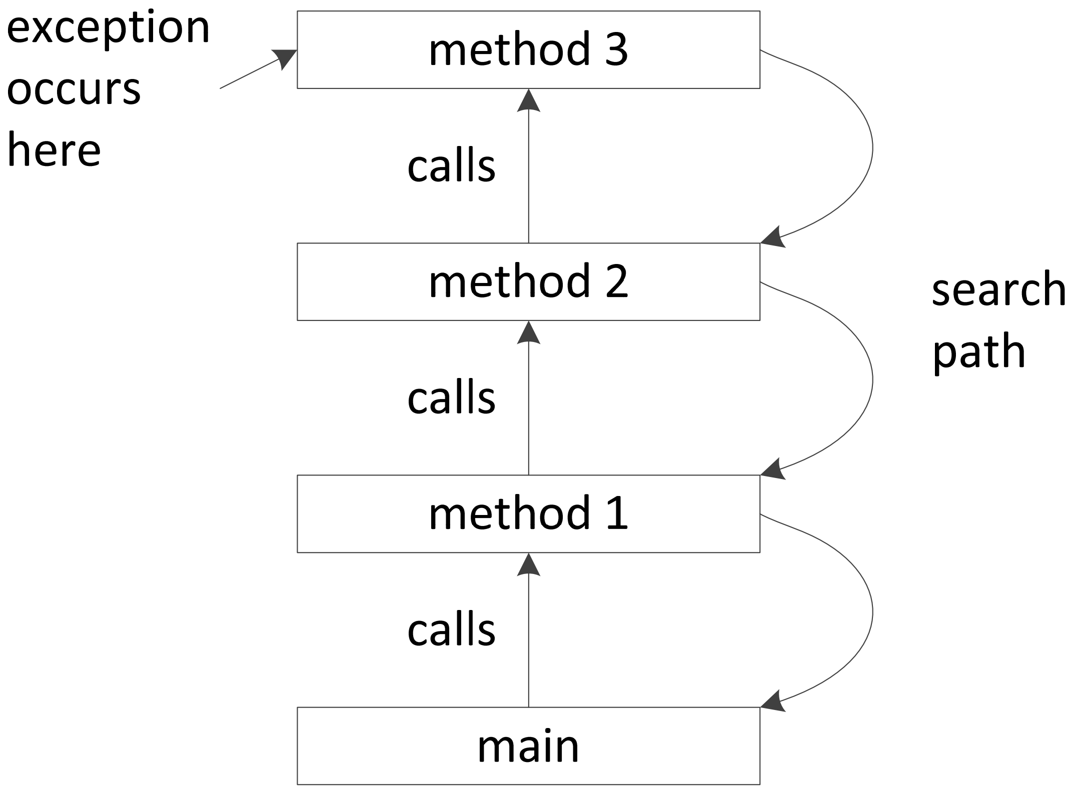
- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields.

# Exceptions

- An *exception*, shorthand for *exceptional event*, is an event that occurs during the execution of a program
- When an exception occurs
  - an exception is *thrown*
  - the runtime system finds an *exception handler*
  - the code in the handler is executed

# Exceptions

- Call stack and exception handler search path



# Exceptions

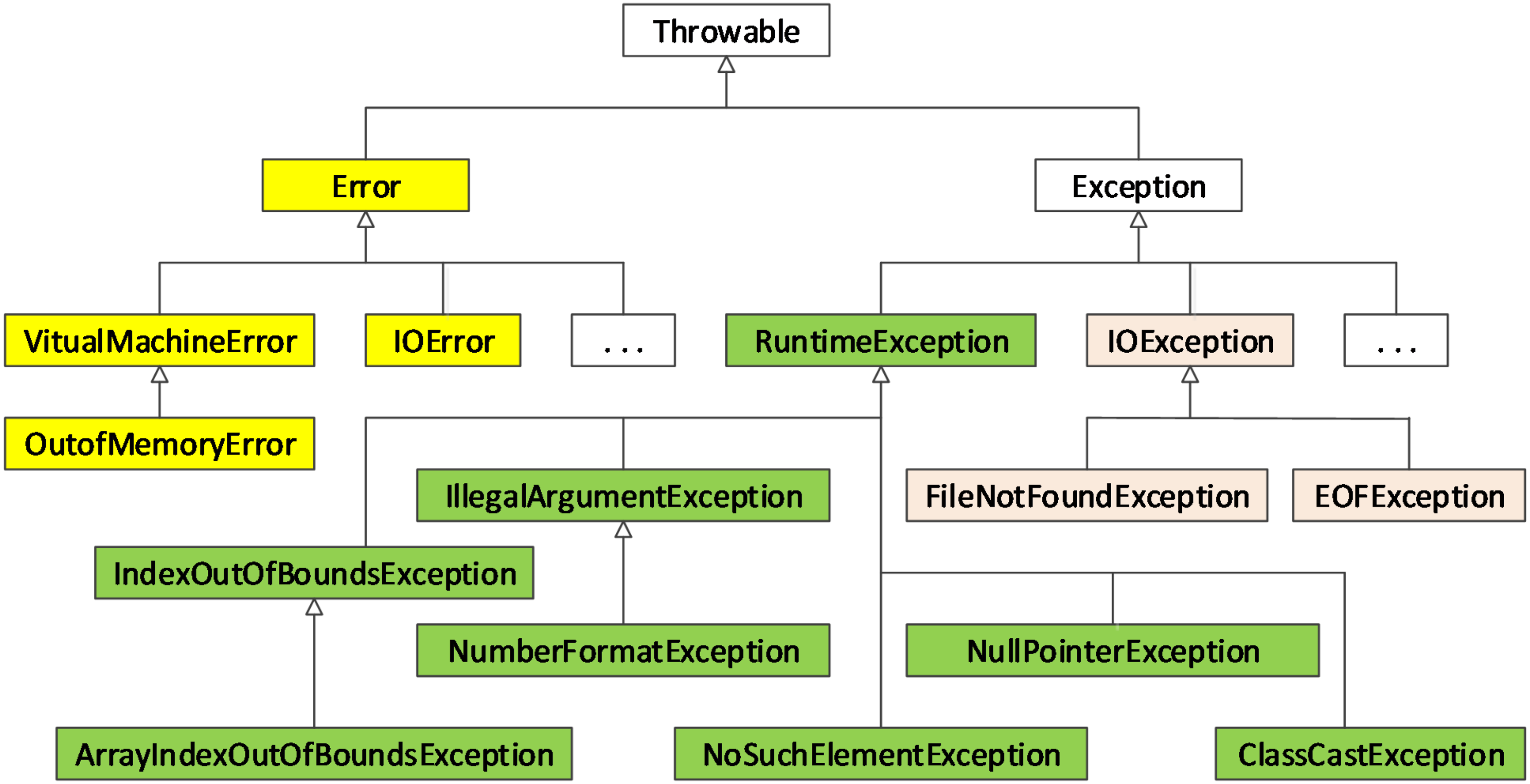
- Try-catch statement

```
try {
            guardedBody
        } catch (exceptionType$_1$ variable$_1$) {
            remedyBody$_1$
        } catch (exceptionType$_2$ variable$_2$) {
            remedyBody$_2$
        } . . .
        . . .
```

- Example: ExceptionDemo.java

# Exceptions

- Java Exception Hierarchy (part)



# Exceptions

- *Errors* (yellow):
  - exception objects of the *Error* class and all of its subclasses.
  - external to the application and they are thrown by JVM.
- *Exceptions*
  - RuntimeException (green)
  - Other exceptions (tan)

# Exceptions

- *Errors*:
  - exception objects of the *Error* class and all of its subclasses.
  - external to the application and they are thrown by JVM.
- *Runtime exceptions* (unchecked exceptions):
  - exception objects of the *RuntimeException* class and all of its subclasses.
  - exceptional events internal to the application, and occur due to mistakes in programming logic
- *Other exceptions* (checked exceptions):
  - all other exceptions
  - If a code may throw a *checked exception*, then it must be in a *try-catch* statement or it must be in a method which is declared with a *throws* clause.

# Generics

- Types are declared using generic names:

```
public class GenericQueue<E> {
        private java.util.ArrayList<E> list = new java.util.ArrayList<>();
        public void enqueue(E e){
        }
        public E dequeue(){
        }
        . . .
}
```

- Instantiated using actual types:

```
GenericQueue<Integer> integerQueue = new GenericQueue<>();
GenericQueue<String> stringQueue = new GenericQueue<>();
```

# Generics

- Generic class definition

```
public class Pair<A,B> {
    A first;
    B second;
    public Pair(A a, B b) {                 // constructor
        first = a;
        second = b;
    }
    public A getFirst() { return first; }
    public B getSecond() { return second;}
}
```

- Instantiation

```
Pair<String, Double> bid;          // declare
bid = new Pair<>(“pi”, 3.14);    // instantiate
```

# Generics

- Generics and arrays

  - Case 1: We have a generic class with parameterized types. We want to declare, outside of the generic class, an array storing objects of the generic class with actual type parameters.

  - Case 2: We have a generic class with parameterized types. We want to declare, as an instance variable of the class, an array storing objects of one of the formal parameter types.

# Generics

- Generics and arrays – Case 1

```
Pair<String, Double>[ ] holdings; // declaring with actual type
                                  // type parameters are allowed
holdings = new Pair<String, Double>[25]; // illegal
holdings = new Pair[25];          // this is allowed
holdings[0] = new Pair<>(“ORCL”, 3.14); // this is legal
```

# Generics

- Generics and arrays – Case 2

```
public class Portfolio<T> {
    T[ ] data;
    public Portfolio(int capacity) {
       data = new T[capacity];              // illegal; compiler error
       data = (T[ ]) new Object[capacity];   // legal, but compiler
                                             // warning
    }
    public T get(int index) { return data[index]; }
    public void set(int index, T element) { data[index] = element; }
}
```

# Generics

- Generic method

```
public class GenericDemo {
    public static <T> void reverse(T[ ] data) {
        int low = 0, high = data.length - 1;
        while (low < high) {           // swap data[low] and data[high]
            T temp = data[low];
            data[low++] = data[high];      // post-increment of low
            data[high--] = temp;           // post-decrement of high
        }
    }
}
```

# Generics

- Generic method

```
String[ ] names = new String[ ]{"john", "susan", "molly"};
GenericDemo.reverse(names);

Integer[ ] integers = new Integer[ ]{10, 20, 30, 40, 50};
GenericDemo.reverse(integers);

Character[ ] chars = new Character[ ]{'a', 'b', 'c', 'd', 'e'};
GenericDemo.reverse(chars);
```

# Generics

- Demonstration
  - GenericQueue.java
  - GenericDemo1.java
  - GenericDemo2.java
  - GenericDemo3.java

# Nested Class

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

# Nested Class

- We use nested classes for the following reasons:
  - *NestedClass* is used only for the *OuterClass*.
  - We want to declare members of the *OuterClass* as private but, at the same time, we want *a* smaller class to be able to access members of the *OuterClass*.
  - We want to implement a data structure which has another smaller data structure as its member.

- The code becomes more readable and it is easy to maintain.
- Nested classes also help reduce name conflict.

# References

- M.T. Goodrich, R. Tamassia, and M.H. Goldwasser, “Data Structures and Algorithms in Java,” Sixth Edition, Wiley, 2014.
- Oracle documentation (https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html)