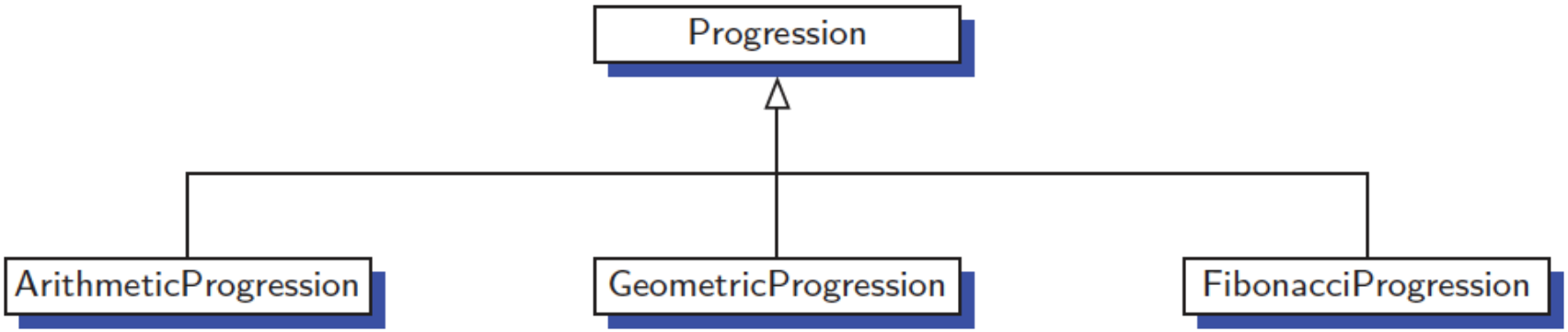
Data Structures and Algorithms

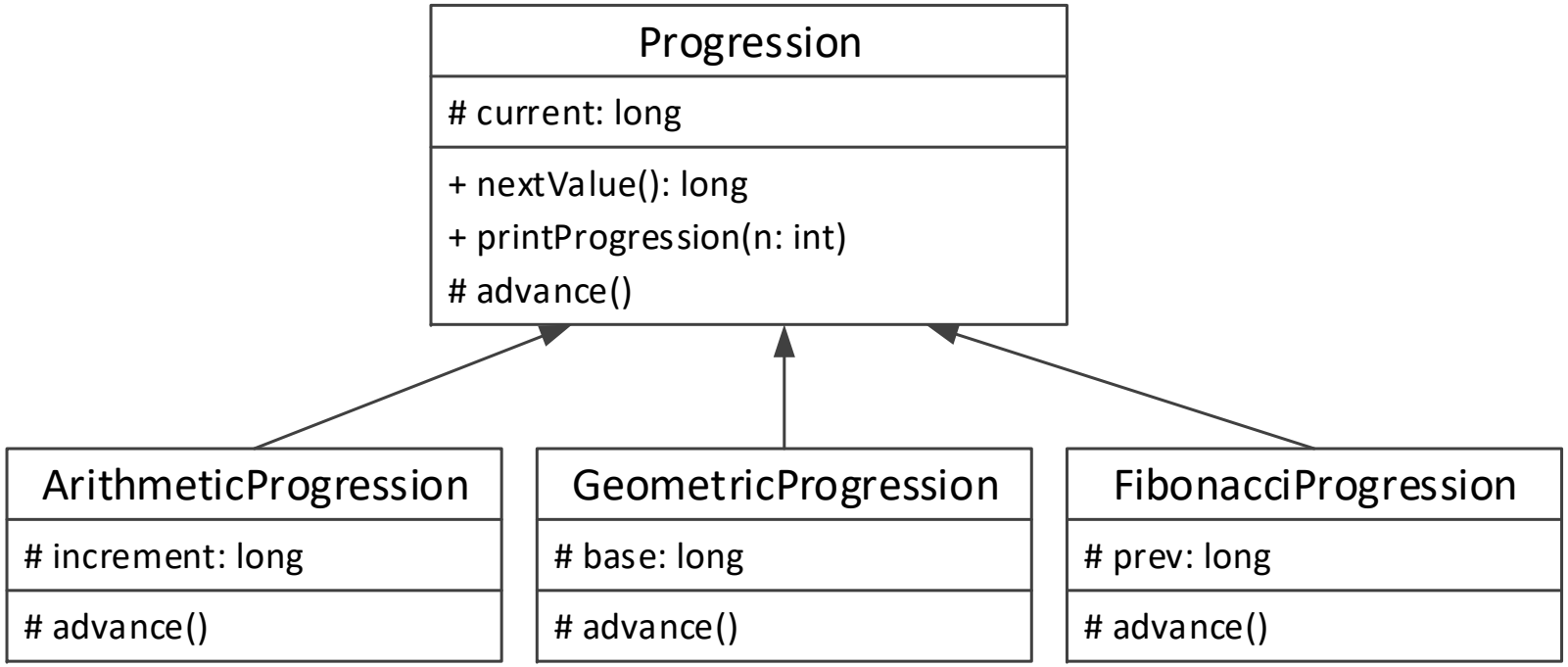
Chapter 2

Inheritance

- Inheritance hierarchy example



Inheritance



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 implement all methods.

Interface

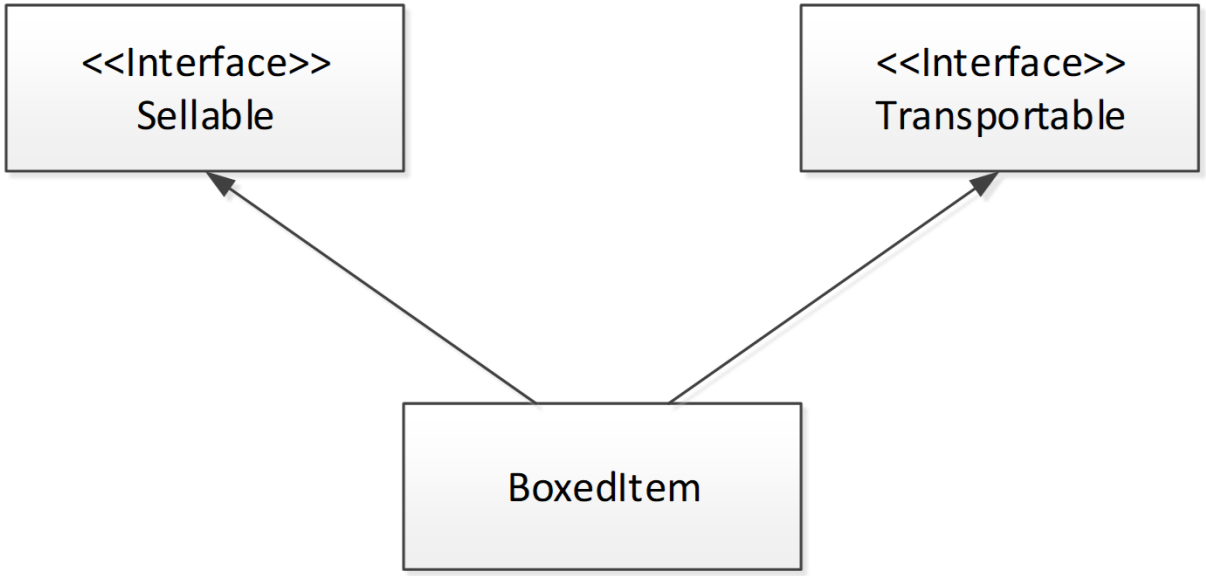
```
1  /** Interface for objects that can be sold. */
2  public interface Sellable {
3    /** Returns a description of the object. */
4    public String description( );
5    /** Returns the list price in cents. */
6    public int listPrice( );
7    /** Returns the lowest price in cents we will accept. */
8    public int lowestPrice( );
9  }
```

Interface

```
1  /** Class for photographs that can be sold. */
2  public class Photograph implements Sellable {
3    private String descript;                    // description of this photo
4    private int price;                          // the price we are setting
5    private boolean color;                      // true if photo is in color
6    public Photograph(String desc, int p, boolean c) {  // constructor
7      descript = desc;
8      price = p;
9      color = c;
10   }
11   public String description( ) { return descript; }
12   public int listPrice( ) { return price; }
13   public int lowestPrice( ) { return price/2; }
14   public boolean isColor( ) { return color; }
15 }
```

Interface

- Multiple inheritance (refer to textbook for more details)

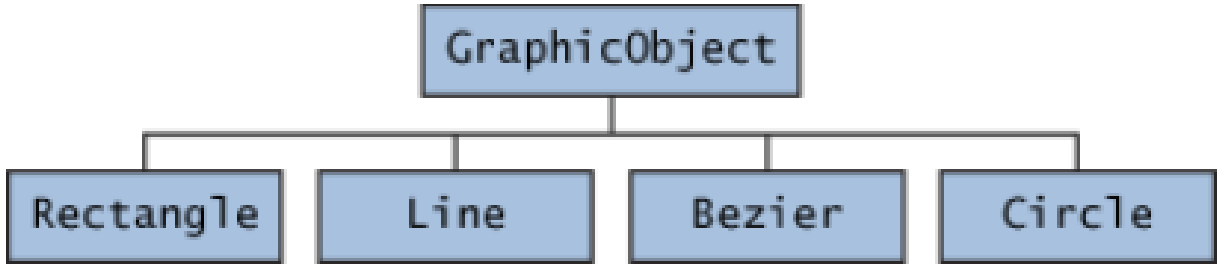


Abstract Class

- 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/landl/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

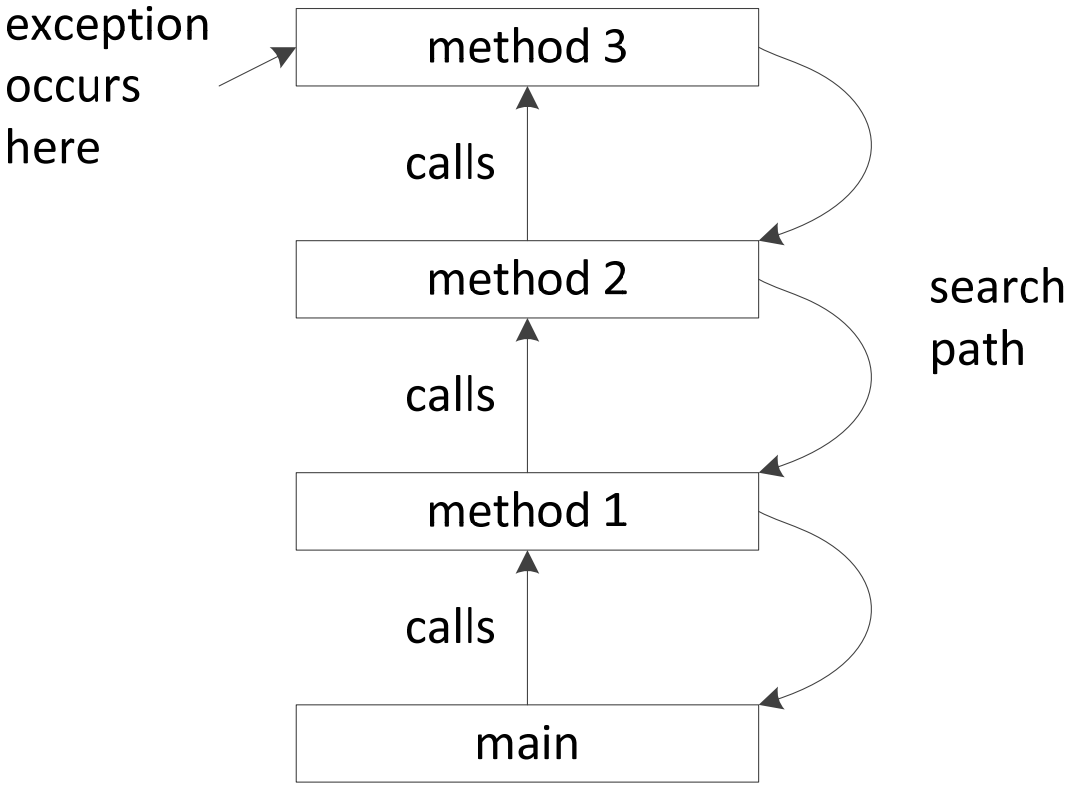
- Consider using abstract classes if any of these statements apply to your situation:
  - You want to share code among several closely related classes.
  - You expect that classes that extend your abstract class have many common methods or fields.

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

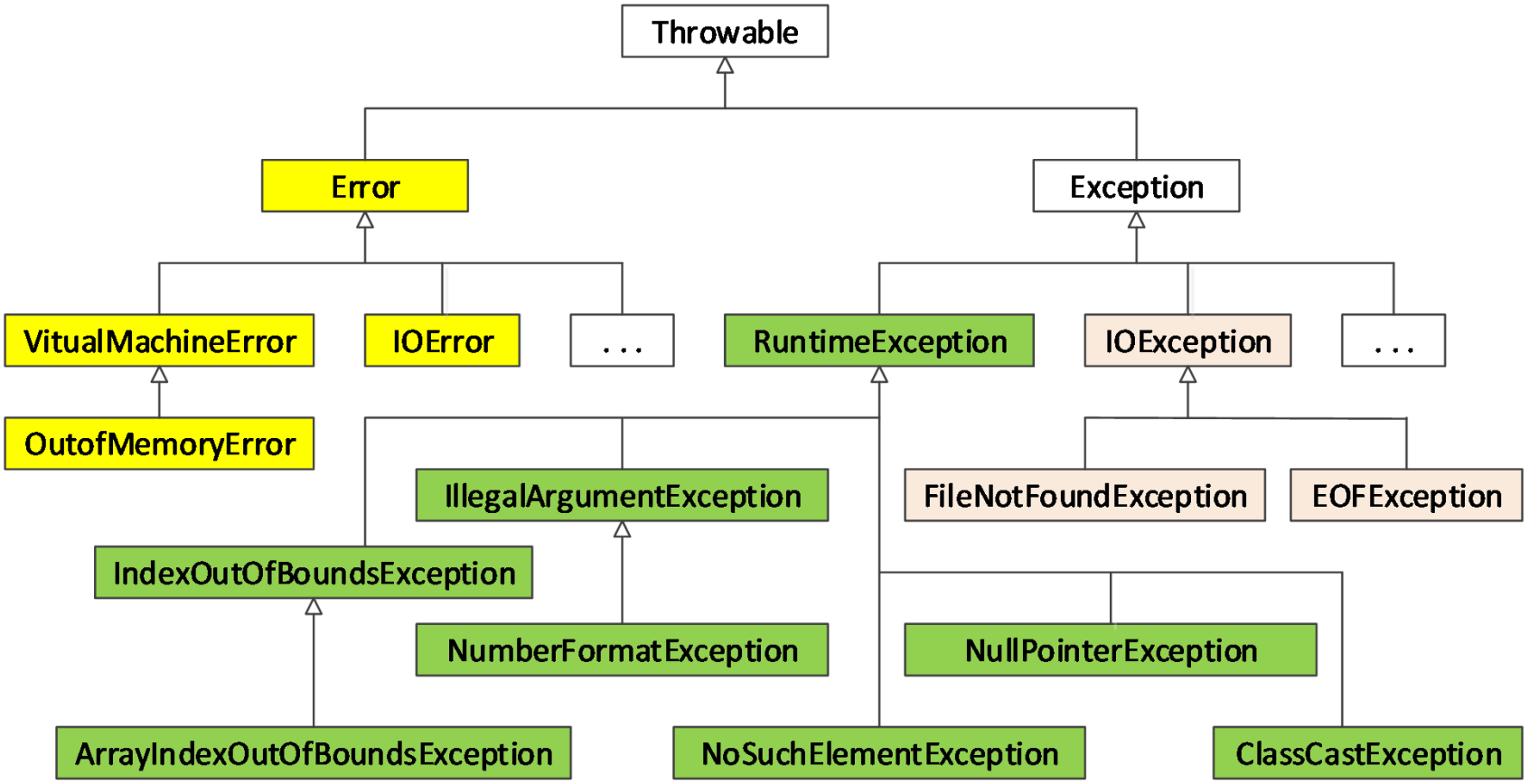
- Try-catch statement

```
try {
        guardedBody
    } catch (exceptionType1 variable1) {
        remedyBody1
    } catch (exceptionType2 variable2) {
        remedyBody2
    } . . .
    . . .
```

- 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.
- *Runtime exceptions* (green, 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* (pink, 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

```
1  public class Pair<A,B> {
2      A first;
3      B second;
4      public Pair(A a, B b) {                  // constructor
5          first = a;
6          second = b;
7      }
8      public A getFirst() { return first; }
9      public B getSecond() { return second;}
10 }
```

- Instantiation

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

Generics

- Generic method

```
1  public class GenericDemo {
2      public static <T> void reverse(T[ ] data) {
3          int low = 0, high = data.length - 1;
4          while (low < high) {          // swap data[low] and data[high]
5              T temp = data[low];
6              data[low++] = data[high];       // post-increment of low
7              data[high--] = temp;            // post-decrement of high
8          }
9    }
10 }
```

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)