

Presentation for use with the textbook **Data Structures and Algorithms in Java, 6<sup>th</sup> edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

# Object-Oriented Programming



# Terminology

- ❑ Each **object** created in a program is an **instance** of a **class**.
- ❑ Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- ❑ The class definition typically specifies **instance variables**, also known as **data members**, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

# Goals

- Robustness

- We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.

- Adaptability

- Software needs to be able to evolve over time in response to changing conditions in its environment.

- Reusability

- The same code should be usable as a component of different systems in various applications.

# Abstract Data Types

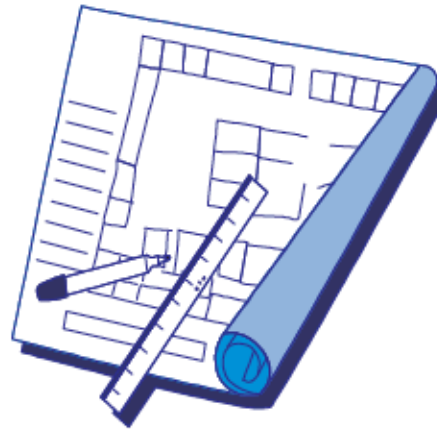
- ❑ **Abstraction** is to distill a system to its most fundamental parts.
- ❑ Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- ❑ An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- ❑ An ADT specifies what each operation does, but not how it does it.
- ❑ The collective set of behaviors supported by an ADT is its **public interface**.

# Object-Oriented Design Principles

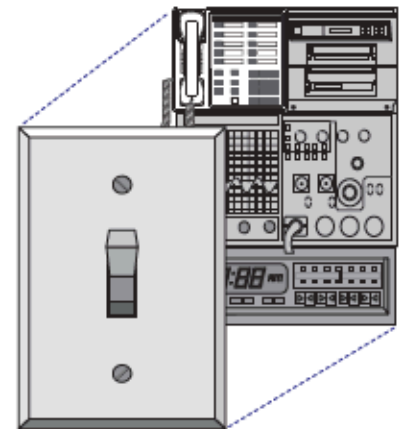
- ❑ Modularity
- ❑ Abstraction
- ❑ Encapsulation



Modularity



Abstraction



Encapsulation

# Interfaces and Abstract Classes

- ❑ The main structural element in Java that enforces an application programming interface (API) is an **interface**.
- ❑ An interface is a collection of method declarations with no data and no bodies.
- ❑ Interfaces do not have constructors and they cannot be directly instantiated.
  - When a class **implements** an interface, it must implement all of the methods declared in the interface.
- ❑ An abstract class also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.

# Design Patterns



## ❑ **Algorithmic patterns:**

- ❑ Recursion
- ❑ Amortization
- ❑ Divide-and-conquer
- ❑ Prune-and-search
- ❑ Brute force
- ❑ Dynamic programming
- ❑ The greedy method

## ❑ **Software design patterns:**

- ❑ Iterator
- ❑ Adapter
- ❑ Position
- ❑ Composition
- ❑ Template method
- ❑ Locator
- ❑ Factory method

# Object-Oriented Software Design

- ❑ **Responsibilities:** Divide the work into different actors, each with a different responsibility.
- ❑ **Independence:** Define the work for each class to be as independent from other classes as possible.
- ❑ **Behaviors:** Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.



# Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

class:	CreditCard	
fields:	<div>– customer : String</div> <div>– bank : String</div> <div>– account : String</div> <div>– limit : int</div> <div># balance : double</div>	
methods:	<div>+ getCustomer() : String</div> <div>+ getBank() : String</div> <div>+ charge(price : double) : boolean</div> <div>+ makePayment(amount : double)</div> <div>+ getAccount() : String</div> <div>+ getLimit() : int</div> <div>+ getBalance() : double</div>	

# Class Definitions

- ❑ A class serves as the primary means for abstraction in object-oriented programming.
- ❑ In Java, every variable is either a base type or is a reference to an instance of some class.
- ❑ A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- ❑ A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

# Constructors

- ❑ A user can create an instance of a class by using the **new** operator with a method that has the same name as the class.
- ❑ Such a method, known as a **constructor**, has as its responsibility is to establish the state of a newly object with appropriate initial values for its instance variables.

# Inheritance

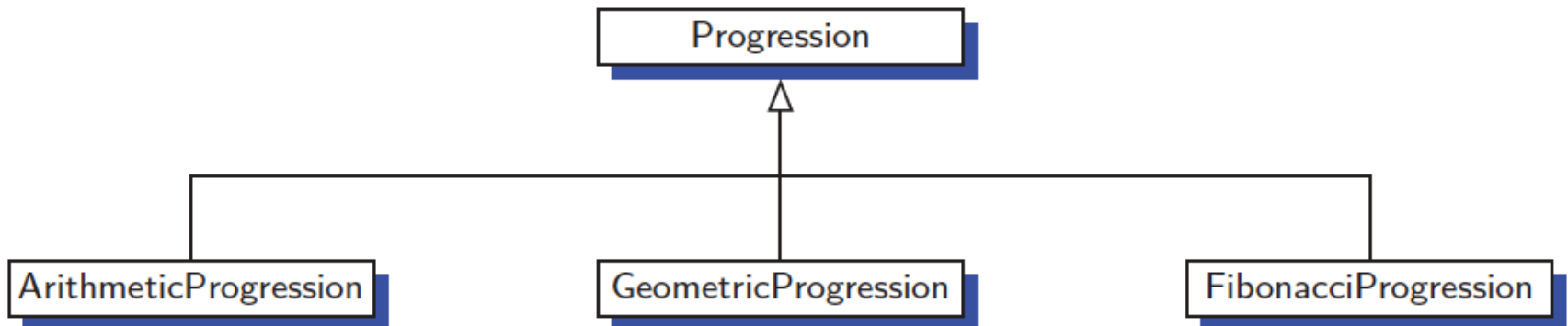
- ❑ A mechanism for a modular and hierarchical organization is **inheritance**.
- ❑ This allows a new class to be defined based upon an existing class as the starting point.
- ❑ The existing class is typically described as the **base class**, parent class, or superclass, while the newly defined class is known as the **subclass** or child class.
- ❑ There are two ways in which a subclass can differentiate itself from its superclass:
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
  - A subclass may also extend its superclass by providing brand new methods.

# Inheritance and Constructors

- ❑ Constructors are never inherited in Java; hence, every class must define a constructor for itself.
  - All of its fields must be properly initialized, including any inherited fields.
- ❑ The first operation within the body of a constructor must be to invoke a constructor of the superclass, which initializes the fields defined in the superclass.
- ❑ A constructor of the superclass is invoked explicitly by using the keyword **super** with appropriate parameters.
- ❑ If a constructor for a subclass does not make an explicit call to **super** or **this** as its first command, then an implicit call to **super( )**, the zero-parameter version of the superclass constructor, will be made.

# An Extended Example

- A **numeric progression** is a sequence of numbers, where each number depends on one or more of the previous numbers.
  - An **arithmetic progression** determines the next number by adding a fixed constant to the previous value.
  - A **geometric progression** determines the next number by multiplying the previous value by a fixed constant.
  - A **Fibonacci progression** uses the formula  $N_{i+1} = N_i + N_{i-1}$



# The Progression Base Class

```
1  /** Generates a simple progression. By default: 0, 1, 2, ... */
2  public class Progression {
3
4      // instance variable
5      protected long current;
6
7      /** Constructs a progression starting at zero. */
8      public Progression() { this(0); }
9
10     /** Constructs a progression with given start value. */
11     public Progression(long start) { current = start; }
12
13     /** Returns the next value of the progression. */
14     public long nextValue() {
15         long answer = current;
16         advance();    // this protected call is responsible for advancing the current value
17         return answer;
18     }
```

# The Progression Base Class, 2

```
19
20  /** Advances the current value to the next value of the progression. */
21  protected void advance() {
22      current++;
23  }
24
25  /** Prints the next n values of the progression, separated by spaces. */
26  public void printProgression(int n) {
27      System.out.print(nextValue());           // print first value without leading space
28      for (int j=1; j < n; j++)
29          System.out.print(" " + nextValue()); // print leading space before others
30      System.out.println();                   // end the line
31  }
32  }
```



# ArithmeticProgression Subclass

```
1 public class ArithmeticProgression extends Progression {
2
3     protected long increment;
4
5     /** Constructs progression 0, 1, 2, ... */
6     public ArithmeticProgression() { this(1, 0); }    // start at 0 with increment of 1
7
8     /** Constructs progression 0, stepsize, 2*stepsize, ... */
9     public ArithmeticProgression(long stepsize) { this(stepsize, 0); }    // start at 0
10
11    /** Constructs arithmetic progression with arbitrary start and increment. */
12    public ArithmeticProgression(long stepsize, long start) {
13        super(start);
14        increment = stepsize;
15    }
16
17    /** Adds the arithmetic increment to the current value. */
18    protected void advance() {
19        current += increment;
20    }
21 }
```

# GeometricProgression Subclass

```
1  public class GeometricProgression extends Progression {
2
3      protected long base;
4
5      /** Constructs progression 1, 2, 4, 8, 16, ... */
6      public GeometricProgression() { this(2, 1); }           // start at 1 with base of 2
7
8      /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
9      public GeometricProgression(long b) { this(b, 1); }     // start at 1
10
11     /** Constructs geometric progression with arbitrary base and start. */
12     public GeometricProgression(long b, long start) {
13         super(start);
14         base = b;
15     }
16
17     /** Multiplies the current value by the geometric base. */
18     protected void advance() {
19         current *= base;                                     // multiply current by the geometric base
20     }
21 }
```

# FibonacciProgression Subclass

```
1 public class FibonacciProgression extends Progression {
2
3     protected long prev;
4
5     /** Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
6     public FibonacciProgression() { this(0, 1); }
7
8     /** Constructs generalized Fibonacci, with give first and second values. */
9     public FibonacciProgression(long first, long second) {
10         super(first);
11         prev = second - first;        // fictitious value preceding the first
12     }
13
14     /** Replaces (prev,current) with (current, current+prev). */
15     protected void advance() {
16         long temp = prev;
17         prev = current;
18         current += temp;
19     }
20 }
```

# Exceptions

- ❑ Exceptions are unexpected events that occur during the execution of a program.
- ❑ An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.
- ❑ In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation.
- ❑ An exception may also be **caught** by a surrounding block of code that “handles” the problem.
- ❑ If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.

# Catching Exceptions

- The general methodology for handling exceptions is a **try-catch** construct in which a guarded fragment of code that might throw an exception is executed.

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

- If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch** block that contains the code to apply an appropriate resolution.
- If no exception occurs in the guarded code, all **catch** blocks are ignored.

# Throwing Exceptions

- ❑ Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.
- ❑ This is done by using the **throw** keyword followed by an instance of the exception type to be thrown.
- ❑ It is often convenient to instantiate an exception object at the time the exception has to be thrown. Thus, a throw statement is typically written as follows:

**throw new** exceptionType(parameters);

where exceptionType is the type of the exception and the parameters are sent to that type's constructor.

# The throws Clause

- ❑ When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method.
- ❑ The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual **throw** statement).
- ❑ For example, the `parseInt` method of the `Integer` class has the following formal signature:

```
public static int parseInt(String s) throws NumberFormatException;
```

# Casting

- ❑ Casting with Objects allows for conversion between classes and subclasses.
- ❑ A **widening conversion** occurs when a type T is converted into a “wider” type U:
  - T and U are class types and U is a superclass of T.
  - T and U are interface types and U is a superinterface of T.
  - T is a class that implements interface U.
- ❑ Example:

```
CreditCard card = new PredatoryCreditCard(...);
```



# Narrowing Conversions

- A **narrowing conversion** occurs when a type T is converted into a “narrower” type S.
  - T and S are class types and S is a subclass of T.
  - T and S are interface types and S is a subinterface of T.
  - T is an interface implemented by class S.
- In general, a narrowing conversion of reference types requires an explicit cast.
- Example:  
`PredatoryCreditCard pc = (PredatoryCreditCard) card;`

# Generics

- ❑ Java includes support for writing generic classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts.
- ❑ The generics framework allows us to define a class in terms of a set of formal type parameters, which can then be used as the declared type for variables, parameters, and return values within the class definition.
- ❑ Those formal type parameters are later specified when using the generic class as a type elsewhere in a program.

# Syntax for Generics

- Types can be declared using generic names:

```
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 }
```

- They are then instantiated using actual types:

```
Pair<String,Double> bid;
```

# Nested Classes

- ❑ Java allows a class definition to be nested inside the definition of another class.
- ❑ The main use for nesting classes is when defining a class that is strongly affiliated with another class.
  - This can help increase encapsulation and reduce undesired name conflicts.
- ❑ Nested classes are a valuable technique when implementing data structures, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure.