OBJECT-ORIENTED PROGRAMMING

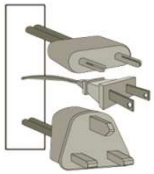
Terminology

- 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 member**s, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

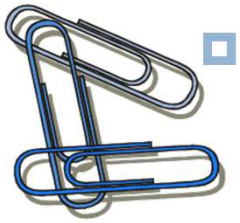
Goals

- 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.

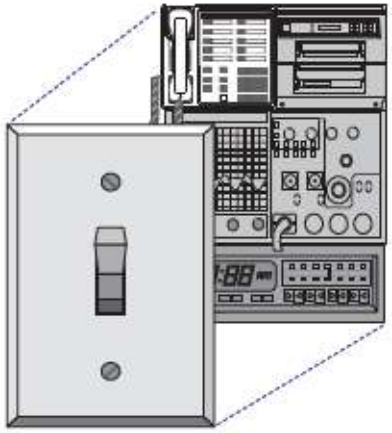


Object-Oriented Design Principles

- Modularity
- Abstraction
- Encapsulation

Modularity

Abstraction



Encapsulation

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**.

Encapsulation

- ***Encapsulation:*** *Different* components of a software system should not reveal the internal details of their respective implementations.
- Encapsulation gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions.
- The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface.
- Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

Modularity

- Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized.
- ***Modularity refers to an organizing principle in which different components*** of a software system are divided into separate functional units.
- Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system.

Design Patterns

- Object-oriented design facilitates **reusable**, **robust**, and **adaptable** software. Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.
- Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. They are called **design patterns, which describe solutions to "typical" software design** problems.
- A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of
  - a name, which identifies the pattern;
  - a context, which describes the scenarios for which this pattern can be applied;
  - a template, which describes how the pattern is applied;
  - and a result, which describes and analyzes what the pattern produces.

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:	− customer : String − bank : String − account : String	− limit : int # balance : double
methods:	+ getCustomer( ) : String + getBank( ) : String + charge(price : double) : boolean + makePayment(amount : double)	+ getAccount( ) : String + getLimit( ) : int + getBalance( ) : double

Class Definitions

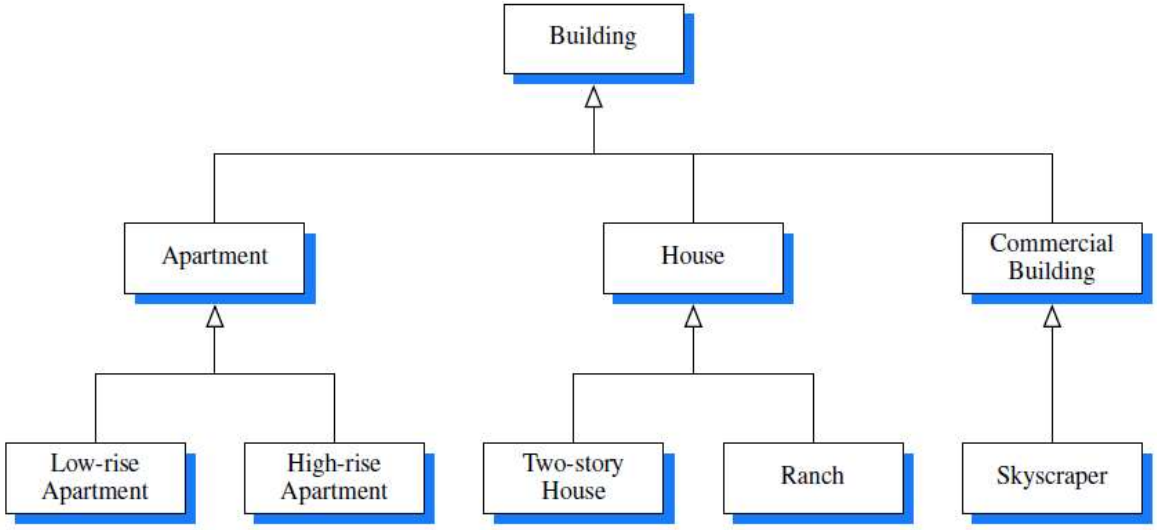
- A class serves as the primary means for abstraction in object-oriented programming.
- In Java, every variable is either a base type or is a reference to an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations 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.



An example of an "is a" hierarchy involving architectural buildings.

Inheritance and Constructors

- 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.
  - 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.

Polymorphism and Dynamic Dispatch

- The word **polymorphism literally means "many forms." In the context of object oriented** design, it refers to the ability of a reference variable to take different forms.
- It indicates the ability of a single variable of a given type to be used to reference objects of different types and to automatically call the method tha is specific to the type of object the variable references.

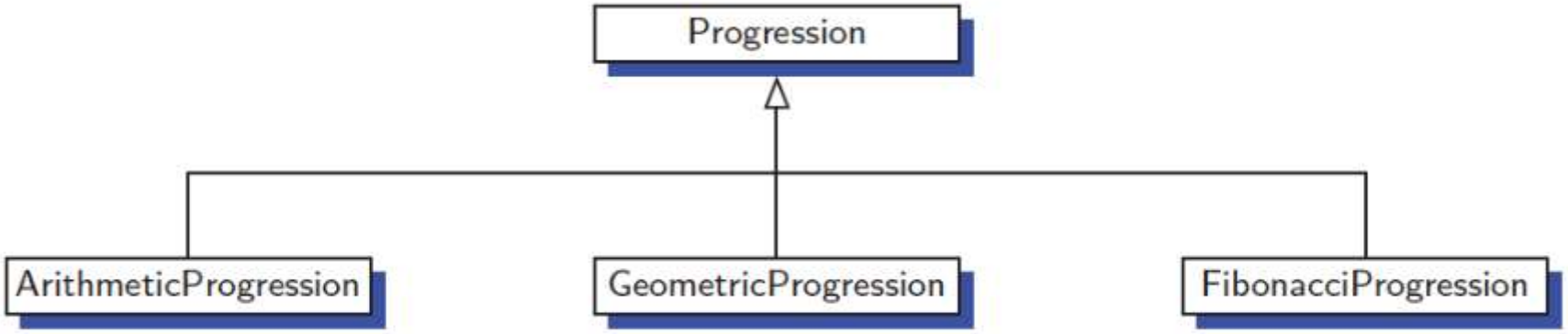
```
class Animal {
  private String type;
  public Animal(String aType) { type = new String(aType);  }
  public String toString() { return "This is a " + type; }
}

class Dog extends Animal {
public Dog(String aType){ super(aType); }
}

class Cat extends Animal {
public Cat(String aName) {
    super("Cat");
    name = aName;
    breed = "Unknown";
  }
  public Cat(String aName, String aBreed) {
    super("Cat");
    name = aName;
    breed = aBreed;
  }
  public String toString() {
    return super.toString() + "\nIt's " + name + " the " + breed;
  }
  private String name;
  private String breed;
}
```

```
public class MainClass {
  public static void main(String[] args) {
    Animal[] theAnimals = { new Dog("Fındık"),
                            new Cat("Minnoş", "Van"),
                             };
    Animal petChoice;

    for (int i = 0; i < 2; i++) {
      petChoice = theAnimals[i];
      System.out.println(petChoice.toString());
    }
  }
}
```

```
This is a Fındık
This is a Cat
It's Minnoş the Van
```

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 $N_{i+1}=N_i+N_{i-1}$



An overview of our hierarchy of progression classes.

The Progression Base Class

```
/** Generates a simple progression. By default: 0, 1, 2, ... */
public class Progression {

  // instance variable
  protected long current;

  /** Constructs a progression starting at zero. */
  public Progression( ) { this(0); }

  /** Constructs a progression with given start value. */
  public Progression(long start) { current = start; }

  /** Returns the next value of the progression. */
  public long nextValue( ) {
    long answer = current;
    advance( );      // this protected call is responsible for advancing the current value
    return answer;
  }
```

The Progression Base Class, 2

```
  /** Advances the current value to the next value of the progression. */
  protected void advance( ) {
    current++;
  }

  /** Prints the next n values of the progression, separated by spaces. */
  public void printProgression(int n) {
    System.out.print(nextValue( ));          // print first value without leading space
    for (int j=1; j < n; j++)
      System.out.print("  " + nextValue( )); // print leading space before others
    System.out.println( );                   // end the line
  }
}
```

ArithmeticProgression Subclass

```
public class ArithmeticProgression extends Progression {

  protected long increment;

  /** Constructs progression 0, 1, 2, ... */
  public ArithmeticProgression( ) { this(1, 0); }       // start at 0 with increment of 1

  /** Constructs progression 0, stepsize, 2*stepsize, ... */
  public ArithmeticProgression(long stepsize) { this(stepsize, 0); }       // start at 0

  /** Constructs arithmetic progression with arbitrary start and increment. */
  public ArithmeticProgression(long stepsize, long start) {
    super(start);
    increment = stepsize;
  }

  /** Adds the arithmetic increment to the current value. */
  protected void advance( ) {
    current += increment;
  }
}
```

GeometricProgression Subclass

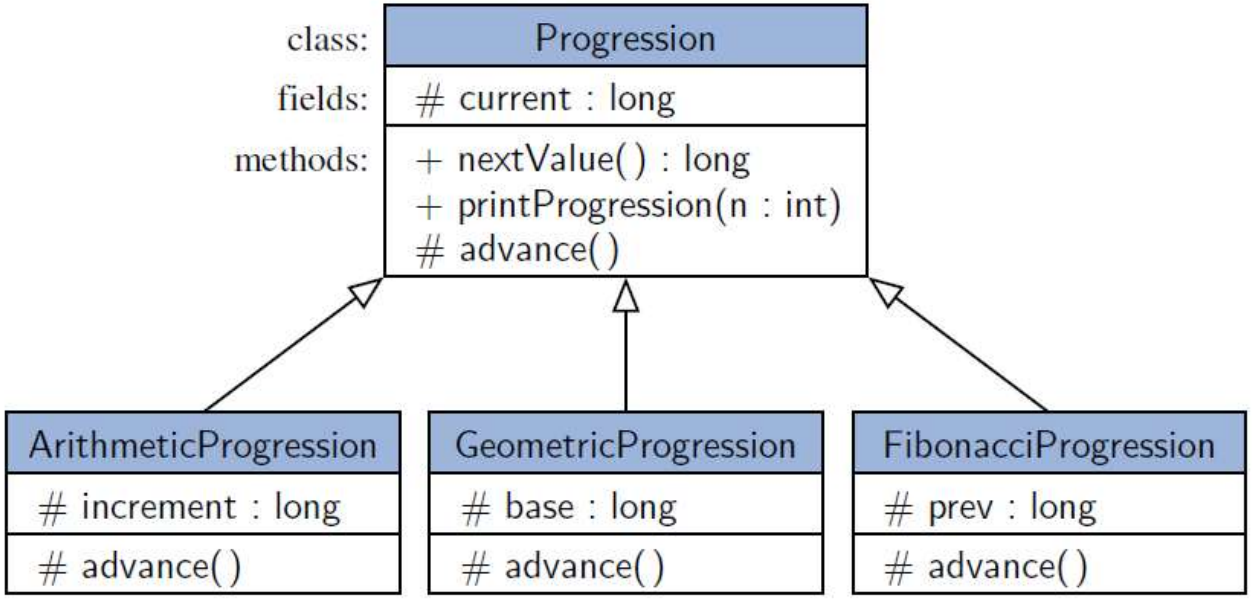
```
public class GeometricProgression extends Progression {

  protected long base;

  /** Constructs progression 1, 2, 4, 8, 16, ... */
  public GeometricProgression( ) { this(2, 1); }          // start at 1 with base of 2

  /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
  public GeometricProgression(long b) { this(b, 1); }                  // start at 1

  /** Constructs geometric progression with arbitrary base and start. */
  public GeometricProgression(long b, long start) {
    super(start);
    base = b;
  }

  /** Multiplies the current value by the geometric base. */
  protected void advance( ) {
    current *= base;                          // multiply current by the geometric base
  }
}
```

FibonacciProgression Subclass

```
public class FibonacciProgression extends Progression {

  protected long prev;

  /** Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
  public FibonacciProgression( ) { this(0, 1); }

  /** Constructs generalized Fibonacci, with give first and second values. */
  public FibonacciProgression(long first, long second) {
    super(first);
    prev = second − first;        // fictitious value preceding the first
  }

  /** Replaces (prev,current) with (current, current+prev). */
  protected void advance( ) {
    long temp = prev;
    prev = current;
    current += temp;
  }
}
```

Progression

- Inheritance diagram for class Progression and its subclasses



```java
/** Test program for the progression hierarchy. */
public class TestProgression {
    public static void main(String[ ] args) {
    Progression prog;
    // test ArithmeticProgression
    System.out.print("Arithmetic progression with default increment: ");
    prog = new ArithmeticProgression( );
    prog.printProgression(10);
    System.out.print("Arithmetic progression with increment 5: ");
    prog = new ArithmeticProgression(5);
    prog.printProgression(10);
    System.out.print("Arithmetic progression with start 2: ");
    prog = new ArithmeticProgression(5, 2);
    prog.printProgression(10);
    // test GeometricProgression
    System.out.print("Geometric progression with default base: ");
    prog = new GeometricProgression( );
    prog.printProgression(10);
    System.out.print("Geometric progression with base 3: ");
    prog = new GeometricProgression(3);
    prog.printProgression(10);
    // test FibonacciProgression
    System.out.print("Fibonacci progression with default start values: ");
    prog = new FibonacciProgression( );
    prog.printProgression(10);
    }
}
```

Interfaces and Abstract Classes

- In order for two objects to interact, they must "know" about the various messages that each will accept, that is, the methods each object supports. To enforce this "knowledge," the object-oriented design paradigm asks that classes specify the ***application programming interface (API), or simply interface, that their objects*** present to other objects.
- The main structural element in Java that enforces an application programming interface 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.

Ex. Interface

```
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( );
}
```

Ex. 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;}
}
```

Multiple Inheritance

- In Java, multiple inheritance is allowed for interfaces but not for classes. The reason for this rule is that interfaces do not define fields or method bodies, yet classes typically do.

Ex. Multiple Inheritance

```
public interface Transportable {
/** Returns the weight in grams. */
public int weight( );

/** Returns whether the object is hazardous. */
public boolean isHazardous( );
}
```

Ex. Interface

```
public class BoxedItem implements Sellable, Transportable {
    @Override
    public int weight() {
        // TODO Auto-generated method stub
        return 0;}
    @Override
    public boolean isHazardous() {
        // TODO Auto-generated method stub
        return false;}
    @Override
    public String description() {
        // TODO Auto-generated method stub
        return null;}
    @Override
    public int listPrice() {
        // TODO Auto-generated method stub
        return 0;}
    @Override
    public int lowestPrice() {
        // TODO Auto-generated method stub
        return 0;}
}
```

Abstract Classes

- In Java, an ***abstract class serves a role somewhat between that of a traditional class*** and that of an interface.
- Like an interface, an abstract class may define signatures for one or more methods without providing an implementation of those method bodies; such methods are known as ***abstract methods.***
- ***However, unlike an interface,*** an abstract class may define one or more fields and any number of methods with implementation (so-called ***concrete methods). An abstract class may also extend*** another class and be extended by further subclasses.

Ex. Abstract Class

```
public abstract class Employee {
    private String name; // Employee name
    private String address; // Employee Address
    private int SSN;  // Social Security Number

    // Abstract method does not have a body
    public abstract double computePay();

    // method with body
    public String getName() { return name; };
}
```

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.

Ex. Exceptions

```
public class ExceptionExample {
    public static final int DEFAULT=5;
    public static void main(String[] args) {
        int n = DEFAULT;
        try {
        n = Integer.parseInt(args[0]);
        if (n <= 0) {
            System.out.println("n must be positive. Using default.");
            n = DEFAULT;
        }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("No argument specified for n. Using default.");
        } catch (NumberFormatException e) {
            System.out.println("Invalid integer argument. Using default.");
        }
    }
}
```

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.

Ex. Throwing Exception

```
public void ensurePositive(int n) {
if (n < 0)
        throw new IllegalArgumentException("That's not positive!");
// ...
}
```

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
{
    // ...
    return 0;
}
```

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:

```
CreditCard card = new PredatoryCreditCard(...); // widening
PredatoryCreditCard pc = (PredatoryCreditCard) card; // narrowing
```

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:

```
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;}
}
```

- 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.

Ex. Nested Class

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.
- Nested classes are divided into two categories: static and non-static.

```
public class CreditCard {
[private] [static] class Transaction {/* details omitted */ }

// instance variable for a CreditCard

Transaction[] history; // keep log of all transactions for
this card

}
```

HW.

- R-2.14 Give an example of a Java code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints the following error message: “Don’t try buffer overflow attacks in Java!”

- Give two examples of Java code frements for Nested Static and Nested Non-Static classes.