Data Abstraction
Object Oriented Programming.

Janyl Jumadinova

September 21–23, 2020
Software Goals

**Robustness:**

In addition to producing the correct output for anticipated inputs, we also want the software to handle unexpected inputs that are not explicitly defined for its application.
Software Goals

Robustness:
In addition to producing the correct output for anticipated inputs, we also want the software to handle unexpected inputs that are not explicitly defined for its application.

Adaptability:
Software needs be able to evolve over time in response to changing conditions in its environment.
Software Goals

**Robustness:**
In addition to producing the correct output for anticipated inputs, we also want the software to handle unexpected inputs that are not explicitly defined for its application.

**Adaptability:**
Software needs be able to evolve over time in response to changing conditions in its environment.

**Reusability:**
The same code should be usable as a component in different systems in varying applications.
How Does OOP support these goals?

Abstraction:
Distill a complicated system down into its most fundamental parts.
How Does OOP support these goals?

**Abstraction:**
Distill a complicated system down into 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:**
Different components of a software system should not reveal the internal details of their respective implementations. Data is accessed through public interfaces.
How Does OOP support these goals?

**Modularity:**
Different components of a software system are divided into separate functional units, which later get integrated into a larger software system.
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.
Interface vs. Abstract Class

implements vs. extends

Collections
Access Modifiers

- **Public**: Everything can access. The class, the package, any subclasses, any external classes.
Access Modifiers

- **Public**: Everything can access. The class, the package, any subclasses, any external classes.

- **Protected**: Everything can access except for external classes.
Access Modifiers

- **Public**: Everything can access. The class, the package, any subclasses, any external classes.
- **Protected**: Everything can access except for external classes.
- **Default / no modifier / “Package-Private”**: Only the class and package can access.
Access Modifiers

- **Public**: Everything can access. The class, the package, any subclasses, any external classes.
- **Protected**: Everything can access except for external classes.
- **Default / no modifier / “Package-Private”**: Only the class and package can access.
- **Private**: Only the class can access.
Inheritance

A mechanism for a modular and hierarchical organization

- 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

“is a” relationship

Building
  △
  Apartment
    △
    Low-rise Apartment
    △
    High-rise Apartment
  △
  House
    △
    Two-story House
    △
    Ranch
  △
  Commercial Building
    △
    Skyscraper
Constructors are never inherited in Java; hence, every class must define a constructor for itself.

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.
Check-in

- Go to itempool.com/jjumadinova/live
- Put your name for today’s participation credit
- Try to answer the questions
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 Extended Example

class Progression

<table>
<thead>
<tr>
<th>fields:</th>
</tr>
</thead>
<tbody>
<tr>
<td># current : long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ nextValue() : long</td>
</tr>
<tr>
<td>+ printProgression(n : int)</td>
</tr>
<tr>
<td># advance()</td>
</tr>
</tbody>
</table>

class ArithmeticProgression

<table>
<thead>
<tr>
<th>fields:</th>
</tr>
</thead>
<tbody>
<tr>
<td># increment : long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td># advance()</td>
</tr>
</tbody>
</table>

class GeometricProgression

<table>
<thead>
<tr>
<th>fields:</th>
</tr>
</thead>
<tbody>
<tr>
<td># base : long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td># advance()</td>
</tr>
</tbody>
</table>

class FibonacciProgression

<table>
<thead>
<tr>
<th>fields:</th>
</tr>
</thead>
<tbody>
<tr>
<td># prev : long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>methods:</th>
</tr>
</thead>
<tbody>
<tr>
<td># advance()</td>
</tr>
</tbody>
</table>

Janyl Jumadinova

Data Abstraction Object Oriented Programming.

September 21–23, 2020
Class Activity 4

Guided Programming: “Progression” example, from Section 2.2.3

1. Accept “activity4” GitHub Assignment.
2. Open “Progression”, “ArithmeticProgression” and “TestProgression” programs.
3. To commit by Wed., Sep. 23: Classes Implemented during class session.
Polymorphism

Polymorphism is the ability of an object to appear and behave differently for the same invocation.

- **Method Overloading**: The notion of having two or more methods in the same class with the same name but different arguments.
  
  ```java
  void foo(int num1);
  void foo(int num1, float num2);
  ```

- **Method Overriding**: Multiple methods with the same arguments, but different implementations.

```java
class Parent {
    void foo (double num) {
        // action
    }
}

class Child extends Parent {
    @Override
    void foo(double num) {
        // overridden.
    }
}
```
Exceptions

- Exceptions are unexpected events that occur during the execution of a program.
- 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

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

```java
throw new exceptionType(parameters);
```

where “exceptionType” is the type of the exception and the parameters are sent to that type’s constructor.
Portion of Java’s Hierarchy of Throwable types
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.
Types can be declared using generic names:

```java
public class Pair<A, B> {
    A first;
    B second;
    public Pair(A a, B b) {
        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.