"OOP is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships."

- Object -- Comprises of:
  - **State** -- object's data
    - controlled (only) by the object
    - externally accessible (only) through its interface
  - **Behavior** -- object's operations (methods)
    - invoked by other objects (interface)
    - invoked internally (implementation)
    - invoke other object's operations
  - **Identity** -- unique name
    - created upon object construction
    - an object reference can be passed between objects

An Object

Your Bicycle

- Objects communicate via message passing, common links:
  - peer-to-peer
Aggregation -- part-of hierarchy.
Messages can span processes and machines with minimal changes to the program!

**Classes**

- Class -- an abstraction of an object, as opposed to concrete objects.
  - represents all *common* characteristics (state and behavior) of its *instances* (objects)
  - useful for design in problem decomposition
  - useful for reuse of common patterns
- A conceptually static entity (although may be accessible at runtime via (non-mutable) objects)
  - Compiler can enforce rules of engagement (e.g., prevent access to non-interface parts, perform type checking)
• Allows to distinguish interface from implementation
  • A class needs to be instantiated:
    o the system allocates memory for instance variables (each instance has its own copy)
    o instance methods are not duplicated
    o all instances share the same copy of a class variable, and class methods can be accessed without instantiation
  • Main class relationships:
    o Inheritance (is-a)
    o Composition (has-a)

The Bicycle Class

- Class vs. type:
  o Class defines internal state and operation implementation
  o type refers only to an object's interface

Inheritance

- Organizes knowledge about the domain (classification)
- Hierarchy often built in both top-down (specialization) and bottom-up (generalization)
- Each sub-class inherits from its super-class(es), and extends/specialized them:
  o new data-members and methods
  o overriding and shadowing existing methods and data-members, respectively
Polymorphism --
- when a request is sent to an object, the particular operation depends on both the request and the receiving object
- Any instance of a sub-class of the specified class can be the receiver -- different invocation requests on different objects may result in different methods invoked.
- benefit: lets clients make few assumptions about other objects -- promotes loose-coupling

Inheritance Issues

Multiple Inheritance: when a class can inherit from multiple super-classes.

- pro - facilitates implementation reuse
- cons - adds complexity and ambiguity
  - more costly dynamic method lookup
  - in case of multiple variables/methods with same name, not clear which one "wins"
Class vs. interface inheritance (subtyping)

- class - defines an object's implementation in terms of another (code sharing)
- interface (abstract class) - describes when an object can be used in place of another
- Java supports single class inheritance but multiple interface inheritance
- 
  programming to an interface  promotes loose coupling, because clients are unaware of the concrete classes that implement them

Inheritance versus Composition

- Inheritance is "white-box" reuse - internals of parent classes are visible to subclasses
  - pros:
    - static (compile-time, supported by the programming language)
    - easy modification of implementation being reused
  - cons:
    - static (can't change inherited implementation at runtime)
    - exposure to parent classes binds sub-classes to parent classes ("Inheritance breaks encapsulation")
    - inherited implementation may not be appropriate
- Composition is "black-box" reuse new functionality by acquiring references to other objects.
  - pros:
    - Objects can be replaced at run-time, so long as they comply with the interfaces
    - fewer implementation dependencies
    - small classes, small hierarchies
  - cons:
    - more objects, more object relationships
    - it is hard to reuse objects "as-is"

- Favoring object composition over class inheritance leads to more reusable and flexible design

**Summary: Benefits of OOP**
Abstraction - A simplified description that suppresses insignificant details - essential for dealing with complex systems through multiple layers, each building on a lower-level implementing the details.
  o object/class supports definition of clear abstraction boundaries

Encapsulation - through objects and distinction between interface and implementation
  o enables to modify implementation w/o affecting callers
    (information hiding)
  o Relieves clients from depending on suppliers' implementation
  o Promotes Modularity - decomposition of program into cohesive and loosely-coupled units
    - Typically require coarser-grained units for "components"
    (e.g., packages)

Reuse - through classes and inheritance

OOP supports well concurrency and distribution! (why ?)

The Name Space and Packages

Advantages:
  o Systematic partitioning of the global name space according to the application's logic.
  o Ability to use the same class name in different contexts.
  o Name collision avoidance.
  o An integral part of Java's security model.

An example: The Java core API — JDK 1.1:
  o The API class library is part of the Java standard.
  o Very reach in functionality.
  o Includes advanced platform-natural GUI support.

The Java Core API

```java
java.applet
java.awt
java.awt.datatransfer
java.awt.event
java.awt.image
java.awt.peer
```
The 'import' and 'package' Keywords

- Specifying to which package should a class belong (done at the beginning of the class source file):

```java
package graphics.shapes;

public class Rectangle {
    ...
}
```

- Instead of writing all the code using full qualified names, classes can be imported as follows:

```java
package graphics.shapes;

import graphics.DrawingArea;
import java.awt.*;
```
public class Rectangle {
    
    DrawingArea.refresh(); // graphics.DrawingArea
    
    Frame drawingFrame = new Frame(); // java.awt.Frame
    
}

Encapsulation Modifiers

- Each field (data or method) is assigned a certain degree of accessibility by either one of the keywords `public` `private` `protected` or the default `package` accessibility.

<table>
<thead>
<tr>
<th>Accessible to:</th>
<th>public</th>
<th>protected</th>
<th>package</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same Class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Class in same package</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Subclass in different package</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Non-subclass in different package</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

- The `public` keyword can also appear before class declaration. It means that the class is accessible anywhere its package is. If it doesn't appear, the class is accessible only in its package.
- Look at the handouts for a table of all Java modifiers.

Exception Handling

- Motivation: Structured handling of exceptional behavior.
- Concepts:
  - An exception — a signal of exceptional condition.
  - To throw an exception — to signal such condition.
To catch an exception — to handle it (take actions).

- An exception object is a subclass of `java.lang.Throwable` that indicates a specific exception. For example:

- Throwing an exception:

  ```java
  if (...) throw new MyException();
  ```

- The try/catch/finally structure:

  ```java
  try {
      ...
  } catch (SomeException e) {
      ...
  } catch (AnotherException e) {
      ...
  } finally {
      ...
  }
  ```

- Exceptions can be left to be caught outside the method. The exceptions thrown by the method are part of its interface:

  ```java
  public void myMethod() throws Exception1, Exception2 {
      // Code that may throw one of the two exceptions.
  }
  ```

- Exception propagation rules:

  (1) Propagate through the lexical block structure of the current method. If the current method is 'main' and the exception was not caught, exit with an error message.

  (2) If not caught, propagate one step through the method call stack and go to (1).
How to use exceptions:
  o Don't over use them. Remember that exceptions should represent non-occasional events and that most of your code should deal with occasional program flow.
  o Don't use them to pass general-purpose data between objects.
  o A lot of though should be given to the design of the exceptions class hierarchy (in big systems) and to when exactly should an exception be caught.
Objects, components, frameworks

by
Haytham Allos

Summary

• Generic forms of modularity
  – Decomposition: Object-oriented programming
  – Assembly: Components
  – Layering: Frameworks
• Shortcomings of OOP
• Benefits of components and frameworks

Forms of modularity

• Decomposition
• Assembly (composition)
• Layering

Mixing

• In actuality, decomposition, assembly, and layering are mixed
• Examples
  – Equipment by assembly
  – Software infrastructure by layering
  – DBMS is component to application
  – Object-oriented programming is decomposition
Object

- Unit of modularity in application programming
  - Chosen to represent, model, or interface real-world objects in the application domain
- Objects interact by calling one another’s methods
  action: parameters → return values
  return action(parameters)
  action(in parameter, out return, inout both)

Real-world objects

What are visible attributes of these objects?
Hidden attributes?
Autonomous behavior?
Reactive behavior?
Interaction?

Encapsulation

Hierarchical decomposition

Containment

Interaction
### Three types of objects

**Proxy**
- Interface to, and act on behalf of the customer within the application

**Customer**
- Predict the behavior of, for purposes of control or coordination

**Heating**
- 

**Modeling**
- 

**Representation**
- 

**Monetary value**
- 

---

### Example

- What are examples of objects that
  - Represent?
  - Model?
  - Proxy?

---

### Proxy objects

- Might also be called interfaces or facades
- Every “foreign” entity should be associated with a proxy object
  - Encapsulate the interaction with that entity
  - Examples: DBMS, CGI interface, user interface
- Proxy object can be thought of as encapsulating the foreign entity

---

### Key ideas

- Align programming methodology with the application context
- Object has tight coupling of data and processing (methods operating on that data)
- Data is encapsulated (available/modifiable only through methods) so
  - Server (not client) is in control
  - Server knows what to do with data, so client doesn’t need to
  - Server can change its implementation

---

### Key ideas (con’t)

- Self-containment (separation of concerns)
  - All operations on data are grouped in one place
- Representation: Structure and interpretation is bundled with the data (becomes information)
- Modeling: Object behaves like real-world object
- Interface: Object represents real-world object
- Representations, models, and interfaces look and behave similarly
Tight vs. loose coupling

<table>
<thead>
<tr>
<th>Objects</th>
<th>DBMS and XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Encapsulates association of processing and data</td>
<td></td>
</tr>
<tr>
<td>• Imposes structure (interacting objects) on application logic, but not data</td>
<td></td>
</tr>
<tr>
<td>• Arbitrary structure and interpretation of data</td>
<td></td>
</tr>
<tr>
<td>• Forces data access and manipulation through SQL interface</td>
<td></td>
</tr>
<tr>
<td>• Imposes structure on data but not application</td>
<td></td>
</tr>
<tr>
<td>• Standardized structure and interpretation of data</td>
<td></td>
</tr>
</tbody>
</table>

Class

- Class is what is in common among a group of similar objects
  - Interface and implementation
  - Economy of expression
  - All objects of same class are instantiations of that class

Example: pocket calculator

Calculator performs the same functions as a pocket calculator

PocketCalculator

- `clear`: -> `void`
- `display`: -> `number_displayed`
- `enter`: `number` -> `number_displayed`
- `plus`: `number` -> `number_displayed`
- `minus`: `number` -> `number_displayed`
- `times`: `number` -> `number_displayed`
- `divide`: `number` -> `number_displayed`
- `equal`: `number` -> `number_displayed`

What are the attributes?

What is the state?

Locating things

- Name
  - Character string, useful for people to remember
- Address
  - Information from which route to thing is specified or can be inferred
  - Available from name service
- Reference
  - Abstract representation of thing: interaction with reference is rerouted to actual entity
  - Especially valuable for things that move

Hierarchical decomposition

- A interacts with B
- A encapsulates B
- Other objects can also have references to B
  - Non-exclusivity
- No other objects possess references to B; only A can interact with B
  - Ownership relationship
### Specialization

- Banana
- Apple
- Strawberry

### Inheritance

- Superclass
- Subclass A
- Subclass B
- Subclass C

### Inheritance (specialization)

- Another economy of expression
- Subclass inherits superclass
- Specialization relationship of classes
- Subclass...
  - Adds state and methods to superclass
  - Possibly modifies existing methods
- Interface inheritance
  - Existing methods not modified

### Inheritance terminology

- Superclass extends or specializes superclass
- Subclass inherits superclass

### Example of inheritance

```java
class Vehicle
  whatis_location()
  whatis_velocity();

class Auto
  fill_radiator();
  replace_pedal();

class Bicycle
  replace_pedal();

class Boat
  bail_water();
```

### Exercise

- What are some objects we see in this room?
- What are their classes?
- What are their attributes and methods?
- How do they illustrate inheritance?
- What are their names, addresses and/or references?
**Polymorphism**

- Subtle but important OOP feature
- Example of abstraction
- Idea:
  - Application deals with superclass
  - Different subclasses substituted at runtime
  - Object substitutes subclass-specific behavior

**Polymorphism example**

```
Ballplayer

Ballplayer joe = new Soccer_player

joe.hit_ball() joe will substitute subclass-specific behavior
```

**Formal definition of polymorphism**

```
method_A()

Superclass defines method_A() (and may or may not implement it)

Subclass subsumes, implements, or modifies method_A()

An object of Subclass can be legally used anywhere Superclass appears
```

**Ballplayer example (con’t)**

```
Ballplayer joe = new Soccer_player

joe.hit_ball() joe will substitute subclass-specific behavior
```

**Question**

- What is an example of using polymorphism in
  - Business application?
  - Social application?
Shortcomings of OOP

• Hierarchical decomposition is informal
• Inheritance problems
  – Implementation inheritance violates modularity (but interface inheritance OK)
  – Multiple inheritance introduces ambiguities
• Reuse problems
  – Insufficient discipline
  – No interface discovery mechanism

OOP shortcomings (con’t)

• Limited interaction modalities
  – Method invocation (request/response)
  – Message (request/no response)
  – No multiplexing and queuing
  – No event notification (publish/subscribe)

Components

• Modules purchased “as is” from outside vendor and used to assemble applications
  – The “industrial revolution” of software
• Emphasize reuse and quality by introducing discipline
• Development methodology much different
  – Must live with available components
  – Scripting and visual assembly languages
  – Integration

Question

• Why has the “industrial revolution” taken so long in software?

Features of components

• Plug and play across
  – applications
  – networks
  – languages
  – operating system platforms
• Much more is standardized
• Richer set of interactions
• Hierarchical decomposition
• Metadata, discovery

Business characteristics of components

• Marketable entity that can be purchased or licensed
  – Binary only
  – Documented, supported
• Not a complete application
• Can be extended and combined with other components
Supercomponents

- Networked, independent entities
  - Security
  - License enforcement, metering
  - Life cycle management
  - Event notification
  - Interface discovery, late binding
  - Transaction control
  - Persistence
  - Others

Business objects

- Many businesses have common elements like competitor, customer, purchase order, invoice, payment, etc.
- Business objects encapsulate these entities, including attributes, behavior, and relationships

Frameworks

- A reusable architecture for a vertical application domain
  - Pre-determined interconnection infrastructure
  - Coordination
  - Framework can be both modified and extended
  - Extension is by adding server objects
- Application layered on framework
- Examples: compound document, ERP

Framework structure

Example framework: compound document

Databases
Databases

- Treat data as a separate asset
  - May be shared by multiple applications
- Provide protection and integrity features appropriate to mission-critical data
  - Access control
  - Integrity constraints
  - Persistence
  - etc.

Relational table

<table>
<thead>
<tr>
<th>Employee</th>
<th>Name</th>
<th>Address</th>
<th>Dept</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Records, rows

<p>| Fields, columns, attributes |
|-----------------------------|---|---|---|</p>
<table>
<thead>
<tr>
<th>Year</th>
<th>City</th>
<th>Accommodation</th>
<th>Tourists</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Oakley</td>
<td>Bed &amp; Breakfast</td>
<td>14</td>
</tr>
<tr>
<td>2002</td>
<td>Oakley</td>
<td>Resort</td>
<td>190</td>
</tr>
<tr>
<td>2002</td>
<td>Oakland</td>
<td>Bed &amp; Breakfast</td>
<td>340</td>
</tr>
<tr>
<td>2002</td>
<td>Oakland</td>
<td>Resort</td>
<td>230</td>
</tr>
</tbody>
</table>
| 2002 | Berkeley| Camping     | 120000 |}

SQL interface

- Presents single abstract interface to the application logic
- Standardized, not vendor specific
  - “Stored procedures” and “extensions” violate this
- Encapsulates various internal details
  - Data partitioning and replication
  - Host mapping
  - File representation
  - etc.

Record at a time processing

- Entries are simple data types or compositions of those types
- Integrity constraints
**Object/table correspondence**

- **Class implementation**
  - Record-at-a-time program
  - Object instance data
  - Attribute

**Class**
- Employee
  - Name
  - Address
  - Dept

**Object attributes**
- Rows can be considered object instances with the same attributes
- Restriction to simple data types
- No methods

---

**Object-relational database**

- A column can store object instances of a given class rather than data of a given simple or compound data type
- Rows can be considered object instances with the same attributes
- Restriction to simple data types
- No methods
- Many other good ideas

---

**Benefits of ORDBMS**

- Extension: manage arbitrarily complex data types
- Migration: preserve and extend existing databases
- Preserve SQL interface
  - OR extensions in latest standard
- All the benefits/experience of earlier databases
  - Access control, data integrity, persistence, etc.
- Killer app: Behind Web/CGI
  - Images, video, audio, animation, applets, etc.