Object-Oriented Programming

Programming Languages

William Killian

Millersville University

Outline

- Terms
- Abstraction
    - Data Encapsulation & Hiding
    - Abstract Data Types
    - Constructors and Destructors
    - Accessors and Mutators
    - Naming Encapsulations
- Inheritance
- Polymorphism
- Design Issues

Terms

Abstraction

A view/representation of an entity that includes only the most significant attributes. An abstraction is always some reasonable subset of information and behavior.

Inheritance

Defining new classes as extensions to existing ones.

Polymorphism

Ability to automatically dispatch to different code based on a shared abstraction and inheritance hierarchy via name.

Terms

Abstract Data Type (Class)

Defined by a set of values and well-defined operations

Object

An instance of an abstract data type (or class)

Method

The name of a behavior/operation defined by a class

Terms

Subclass / Derived Class

A class than inherits from another

Superclass / Parent Class

A class that is inherited by another

Abstraction

Abstraction

Foundational to programming languages

Two levels:

- Process abstraction (via subprograms)
- Data abstraction (via records, abstract data types)

Abstract Data Type:

User-defined data type that satisfies two conditions:

1. The representation is hidden from the users of the type. Only explicit operations provided in the definition are usable.
2. The declarations of the methods and supported operations are contained within a single programming structure.

Abstract Data Type Requirements

The representation is hidden from the users of the type. Only explicit operations provided in the definition are usable.

- Reliable – hiding the data representation makes it harder for others to change the representation in an undefined way.
- Modular – the representation can be changed without changing the user/client code
- Readable – name conflicts are less likely in an "owned" code model

Abstract Data Type Requirements

The declarations of the methods and supported operations are contained within a single programming structure.

- Modularity is key
- A method of program organization and structure
- Separate compilation / analysis
    - For massive projects, this is crucial

Data Encapsulation

- All information is stored within a single entity/unit
- Not *all* information needs to be publicly visible
- State of a structure can be safety updated/operated on

```
class Player
    hp : int
    name : string
    position : vec3
```

Data/Information Hiding

- Should be able to hide any data or internal operation from a client/user
- Introduce *visibility modifiers* to the language

`public`

information is publicly accessible

`private`

information is only accessible within the class

`protected`

private + visible to derived classes

Object Creation: Constructors

- Special functions that allow the user to initialize the data members of class instances
- Can include parameters to provide initial control/parameterization of the objects
- Implicitly called upon creation of an object
- Can also be explicitly called!
- Name == class name

Object Destruction: Destructors

- Special function that cleans up an instance when its destroyed.
  - Usually only needed if heap-dynamic storage was allocated or used within an instance’s lifetime
- Implicitly called when the instance’s lifetime ends
- Can also be explicitly called!
- Only exists in languages that require manual memory management

Object Access: Accessor Methods

- Functions that belong to a class where information is only accessed (never modified)
- Often named as **noun()** or **getNoun()**
- Immutability: instance variables must not change when calling (violation of contract)
- Some languages allow us to force the contract of zero instance modification (**const** in C++)

Object Mutation: Mutator Methods

- Functions that belong to a class where the state is expected to evolve or change in some way
- Often named as **verb()** e.g. push_back
- Often will require parameters
- Mutable: one or more instance variables will be changed. The “state” of the program evolves as an instance of a class changes.

Object Properties

- Enable a programmer to specify a special code path for entities that would normally have three parts:

• private data	`name`
• public accessor method	`getName`
• public mutator method	`setName(name: string)`

- Often used to condense <u>*validation code*</u> to a single area
- Can create “artificial” entities that correspond to data-backed members

Object Properties in C#

```
public double Hours {
    get {
        return seconds / 3600;
    }
    set {
        if (value < 0 || value > 24) {
            seconds = value * 3600;
        } else {
            throw new ArgumentException(…);
        }
    }
}
```

Encapsulations

- Large programs have two primary needs
  1. Some foundational means of organization
  2. Some means of *partial compilation* (programming units that are smaller than the entire program)

- Solution:
  - Group a collection of subprograms together (often logically related) and compile them as a unit
  - This group is called a compilation unit / encapsulation

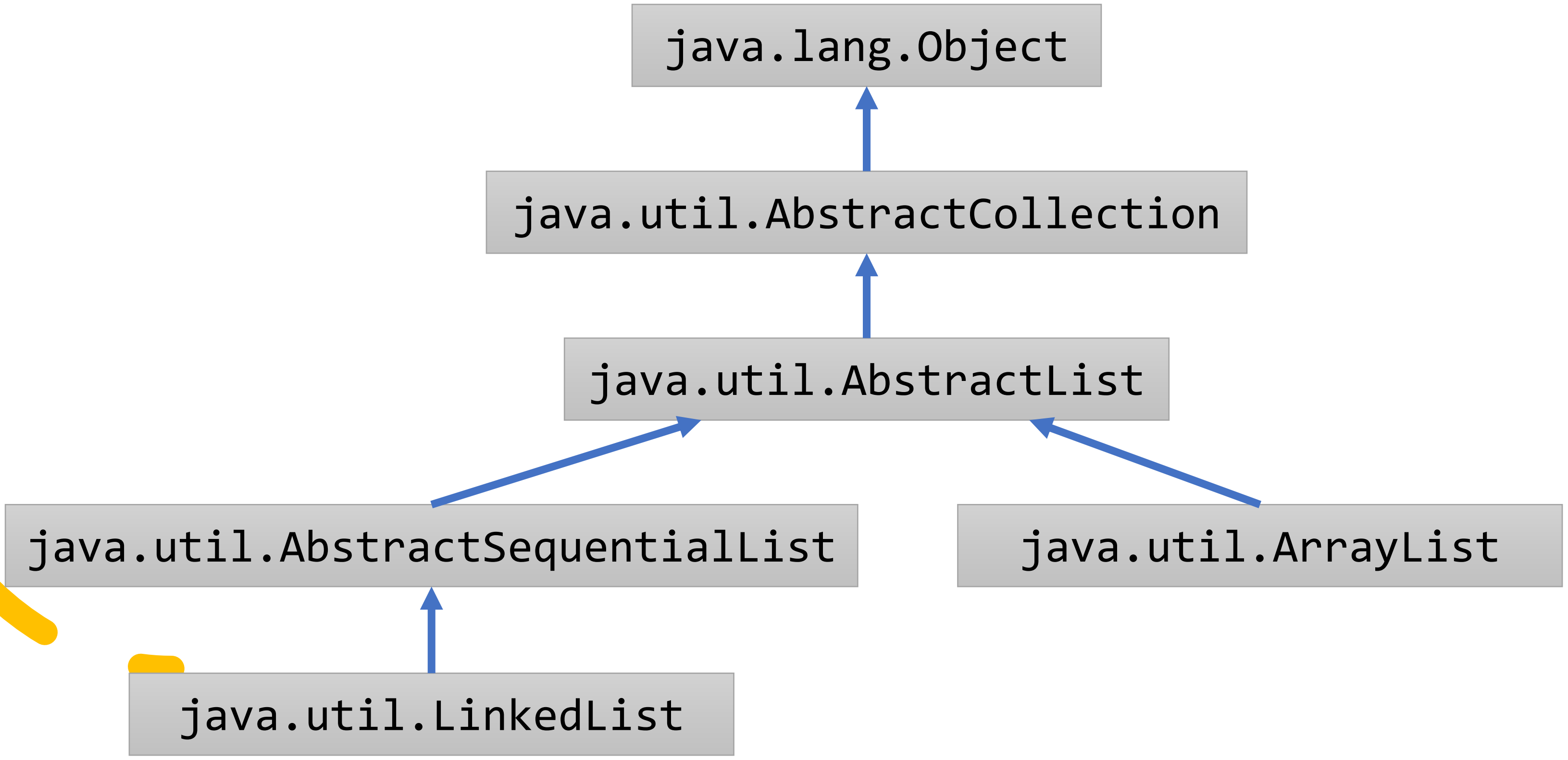
Naming Encapsulations

- Often need to group related or similar entities into some encapsulation by a name
    - Naming things is hard!
    - We could have two different entities with a type of ***Vector***
- Different Languages solve this in different ways:
    - C++ – namespaces (and modules in C++20)
        - Everything in the std namespace is owned by the C++ Standard
        - Everything in the glm:: namespace is maintained by the glm library
    - Java – packages
        - java.* is part of the java standard library
        - javax.* is part of the java eXetensions to the standard library
    - C# – namespaces
        - Like namespaces (System is a built-in namespace)
    - Ruby – modules

Inheritance

Class Hierarchy

A class hierarchy is a graphical representation of the relationship of one class to all other related classes



$$\begin{tikzcd}
 & \text{java.lang.Object} & \\
 & \text{java.util.AbstractCollection} \arrow[u] & \\
 & \text{java.util.AbstractList} \arrow[u] & \\
\text{java.util.AbstractSequentialList} \arrow[ur] & & \text{java.util.ArrayList} \arrow[ul] \\
\text{java.util.LinkedList} \arrow[u] & &
\end{tikzcd}$$

Subclass

- A class that inherits from another class
- Also known as a derived class
- Special way to access/call a method from the class that we are inheriting from:
  - **`super`** in Java
  - **`base`** in C#
  - `BaseClassName::method` in C++
    - :: is the scope resolution operator

Superclass

- A class that is inherited by another class
- Also known as a base class or parent class
- No way for us to know what classes inherited from us statically or even at runtime (without testing)

Types of Classes

Final or Sealed

A class that can not be a superclass

Interface

A class that provides only method signatures

- no method implementations permitted

- no data members permitted

Abstract Class

A special type of class that can provide data members and method implementations, but must not provide the implementation for at least one method

Interface (non-instantiable)

```
// Java / C#
public interface Drawable {
  void draw()
}

// C++
class Drawable {
public:
  virtual void draw() = 0;
}
```

Abstract Class (non-instantiable)

```
// Java / C#
public abstract class Shape {
  public Color color;
  public abstract void draw();
}
// C++
class Shape {
public:
  Color color;
  virtual void draw() = 0;
}
```

Inheritance

Java

- Interface inheriting interface: `extends`
- Class inheriting class: `extends`
- Class “inheriting” interface”: `implements`

```
class DrawableSquare extends Square implements IDrawable
```

C#

- No keyword, just a colon

```
class DrawableSquare : Square, IDrawable
```

C++

- No keyword, just an optional visibility modifier & colon

```
class DrawableSquare : public Square, IDrawable
```

Changing Visibility of Inheritance

- Java, C#, and most Object-Oriented Languages do not allow you to change the visibility of inheritance
- C++ does ☺

```
class Base {
  private: int x;
  public:  int y;
}
class Derived1 : public Base {
  public:  int z;
}
class Derived2 : private Base {
  public:  int z;
}
```

Multiple Inheritance

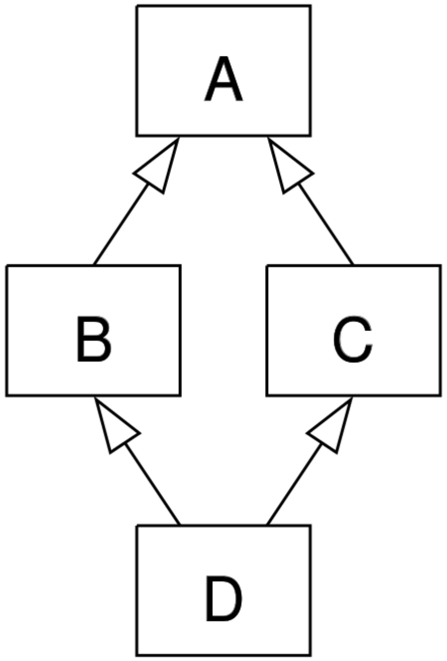
- Java, C#, and most Object-Oriented Languages do not allow you to have multiple inheritance
- C++ does ☺

```
class Drawable {
  public:  vec3 color, position;
}
class Rectangle {
  public:  int width, height;
}
class DrawableRectangle :
  public Drawable, public Rectangle {
}
```

Multiple Inheritance

- Problematic due to the “Diamond Problem”
- Also known as the ***Diamond of Death***

  - If I call a method from A in D, which “path” do I take?
    - D -> B -> A
    - D -> C -> A
  - Especially problematic of B or C ***overload*** that particular method



Overloading / Overriding

- When we provide our own definition for a method that already exists.
- For example:
  - a `Circle` class has a `draw()` method
  - a `FilledCircle` class should have its `draw()` method behave/act differently
  - `FilledCircle` otherwise is identical to `Circle`
  - Therefore, `FilledCircle` should ***overload*** *(or override)* `Circle`’s provided `draw()` method

Parent vs. Subclass Differences

Three primary ways a class can differ from its parent:

1. **[SUBCLASS]**
   Introduce new variables/methods
2. **[SUBCLASS]**
   Modify the behavior of inherited methods
3. **[PARENT]**
   Hide/reduce visibility of variables/methods to make them invisible to the subclass

Class Contents

	Variables (state)	**Methods (behavior)**
Class (static)	• Information shared among all instances of a single class. • Pseudo-global variable	• Behavior/actions that are related to the class, but do not rely on an instance of the class • Java creates functional modules this way (e.g. Arrays, Collections, Math)
Instance (member)	• Information that exists for each instance of a class. • Serves as the data blueprint • Unique, non-shared data	• Behavior/actions that require an instance of the class • Examples: • "draw" the circle • "move" the cursor • "click" the mouse

Polymorphism

Dynamic Binding

- Classes can be defined to contain *polymorphic behavior*
- When we have an instance of a polymorphic object, we should be able to *resolve* the object's actual type and run the right code
- Example:
  - Drawing shapes

Dynamic Binding Example

```
Shape* s = new Rectangle;
Shape* t = new Circle;

s->draw(window);
t->draw(window)

delete s;
s = new Circle;

s->draw(window);

delete s;
delete t;
```

```
class Shape {
  int x, y;
public:
  virtual void draw(Window*) const = 0;
};
class Rectangle : public Shape {
  int width, height;
public:
  void draw(Window* win) const override {
    // draw four lines
  }
};
class Circle : public Shape {
  int radius;
public:
  void draw(Window* win) const override {
    // draw many tangent lines
  }
};
```

Full Example: https://godbolt.org/z/aEWbje

Class Instance Records

So how does Dynamic Binding actually work?

We need to keep track of all the data and *information about a class*

What information do we need?

- *Class data members*
- *Class methods*
- *But we also need to consider the parent's information, too!*

Class Instance Records focus on the ***storage*** of the state of an object

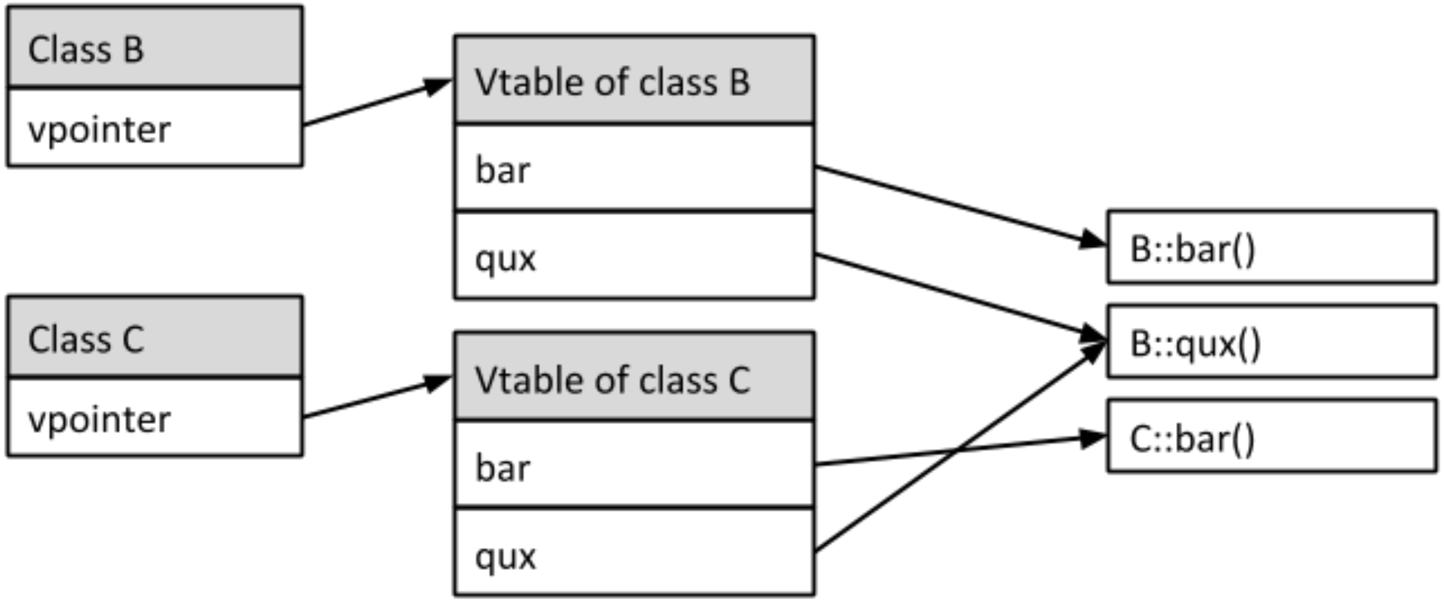
Class Instance Records (CIR)

- Static – constructed at compile time
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Access to all instance variables is done the exact same way as normal records (lookup by name)
  - Efficient
  - Static
- What about methods?

Virtual Tables

- Methods that are **not** “virtual” / “abstract” don’t need anything special done as they will Just Work ™
- But dynamically bound methods must have entries in the Class Instance Record
    - Calls to the methods can be connected to the code with a pointer to the function’s address in the CIR
    - This creates a lookup table of function names to function addresses.
    - We call this table a **virtual method table**, or vtable
    - Method calls are often represented as offsets from the beginning of the vtable

Virtual Tables



```
class B {
public:
  virtual void bar() {}
  virtual void qux() {}
};
```

```
class C : public B {
public:
  virtual void bar() {}
};
```

Reflection

- Runtime access to:
  - Types
  - Structure
  - (called **metadata**)

- Able to *dynamically modify their behavior*

- The process of a program examining its metadata is called *introspection*

Reflection

- Why use reflection?
  - ***Because it's cool***
- Software Tools!
  - Debuggers need to examine private fields
  - Test systems need to know all methods of a class
  - Visual IDEs use type information to assist developers
  - Class browsers need to enumerate all classes of a program
- Downsides:
  - Expensive (performance)
  - Exposes private fields and methods
  - Security

Reflection in Java

- Supported with `java.lang.Class`
- Java runtime instantiates an instance of `Class` for each object in the program!
- There is a `getClass` method for every single object

```
String s = "hello";
Class<?> c = s.getClass();
```

- If there is no object, you can use the class static field

```
Class<?> c = String.class;
```

Reflection in Java

Class has four super useful methods:

- `getMethod` searches for a **public** method
- `getMethods` returns all **public** methods of a class
- `getDeclaredMethod` searches for a method
- `getDeclaredMethods` returns all methods of a class

There are also a couple other useful classes part of reflection: Method (including invoking!) and Field

Demo in jshell

Design Issues

Object Exclusivity

- When designing an OOP language, should everything be an Object?
  - Ruby does this
  - Java does not
  - OCaml does not
- Advantages:
  - *Elegant*
  - Pure
- Disadvantages:
  - Slow operations on simple objects

Object Exclusivity

- Should we add objects to a complete type system?
  - “Strong typing”
  - Java does this
- Should we include a “normal” type system and make everything else objects?

Subtyping

- Are subclasses subtypes?
- Is a Circle a Shape?
    - If a derived class is-a parent class, then object of the derived class must behave **exactly the same** as the parent class

- **Subclasses** inherit <u>*implementation*</u>
- **Subtypes** inherit <u>*interface + behavior*</u>

Multiple Inheritance

- Should we allow multiple inheritance?
  - Recall the diamond of death
- Does it make sense to have / enable it?
- *Advantages*:
  - It's natural
  - Convenient, valuable
- *Disadvantages*:
  - More complex to implement (name collision resolution)
  - Dynamic binding costs more (no penalty for static binding)

Object (De)allocation

- Where do we allocate objects?
    - If they are true ADTs, they should be allocated from anywhere
        - (not the case in C++ -- must be heap-dynamic)
    - If they are heap-dynamic, references can always be modeled as a pointer or reference variable
        - Java does this
    - If objects are stack dynamic, there is a ~~cool~~ ***<u>awful</u>*** problem called *<u>object slicing</u>* that occurs
- Is deallocation explicit or implicit?

Dynamic / Static Binding

- Should all bindings of method calls be dynamic?
  - If none are, you lose the advantage of dynamic dispatch
  - If all are it’s inefficient
- Maybe the design should permit both?
  - Java does this with @Override and abstract/interface
  - C++ does this with virtual / override

Nested Classes

- Should we be able to nest classes?
- If we do, should the class instance have its own class or should it be shared among all classes?
  - Java allows both … either with(out) the static keyword
- What should be made visible to the nested class from the outer class?
  - Private fields/methods?
  - Other nested classes?

Object Initialization

- Are objects initialized to values when they are created?
- Do constructors automatically get called or do we need to manually do so?
- How are parent class members initialized when subclass objects are created?
  - Java: super() delegating constructor
  - C++: Base() delegating constructor as first item in member initializer list