- **Basic OOP Concepts**
- **Introduction**
  - **Goal of OOP: Reduce complexity of software development by keeping details, and especially changes to details, from spreading throughout the entire program.**
  - **This presentation assumes "Basic Class Design" presentation.**
  - **Definitions**
    - *Client Code - the code that uses the classes under discussion.*
    - *Coupling - code in one module depends on code in another module*
      - Change in one forces rewrite (horrible!), or recompile (annoying), of code in the other.
  - **Four key concepts of OOP**
    - *Abstraction - responsibilities (interface) is different from implementation*
    - *Encapsulation - guarantee responsibilities by protecting implementation from interference*
    - *Inheritance - share interface and/or implementation in related classes*
    - *Polymorphism - decouple client from exact class structure*
- **Guidelines for Designing Individual and Concrete Classes**
  - **"Concrete" classes - no inheritance or polymorphism involved.**
  - **Two kinds of classes:**
    - *Objects that are in the problem domain.*
    - *Objects that support the other objects.*
  - **Design a class by choosing a clear set of responsibilities.**
  - **Principle: Avoid heavy-weight, bloated, or "god" classes - prefer clear limited responsibilities.**
  - **Make all member variables private in each class.**
  - **Put in the public interface only the functions that clients can meaningfully use.**
  - **Friend classes and functions are part of the public interface, and belong with the class.**
  - **Make member functions const if they do not modify the logical state of the object.**
- **Guidelines for Class Hierarchies**
  - **The base class must represent a meaningful abstraction in the domain.**
  - **Make base classes abstract - corresponding to the abstraction in the application domain.**
  - **Most-derived ("leaf") classes should represent concrete objects in the domain.**
  - **Intermediate classes should also represent meaningful abstractions in the domain.**
  - **Inherit publicly only to represent the "is-a" (substitutability) relationship.**
  - **Distinguish between the public and protected interface.**
  - **Put common functionality as high up in the class hierarchy as it is meaningful to do so.**
  - **Let each class in a hierarchy be responsible for itself.**
  - **To re-use code that is in a class, prefer using a member variable of that type (aggregation, has-a relationship) instead of private inheritance (implemented-with relationship).**
  - **If the class might be used as a base class, declare a virtual destructor.**

- **The fundamental OOP technique: A polymorphic class hierarchy.**
  - **Fundamental: Use virtual functions instead of "switch on type" logic!**
  - **Use inheritance and polymorphism to hide details and changes.**
    - *Basic concept: Hide present and future differences under a base class, and use polymorphism to get the different behavior controlled by the same client code.*
  - **If some functions are defined only in derived classes, choose a way to access them.**
    - *"The fat interface problem" - no single good label for the problem.*
      - If the classes in an inheritance hierarchy are not uniform in what functions it makes sense for them to have, you can't choose a set of base class virtual functions that are equally applicable to all derived classes.
      - So if some functions are meaningful for only some derived classes, how do we enable the Client to access them?
    - *Four solutions:*
      - **1. "Fat interface"**
      - **2. Use separate containers to keep track of objects by their types.**
      - **3. Downcast safely.**
      - **4. Use a "Capability" base class and "capability queries."**
    - *Best solution depends on details of the situation.*
- **Additional Techniques for OOP**
  - **Decouple client from object creation details with a virtual constructor or factory.**
  - **Consider making the base class an interface class.**
  - **Let Derived class supply additional specialized work to the work done by the Base class.**
  - **A non-virtual member function can call a virtual member function.**
  - **Use dynamic_cast appropriately.**
    - *Flakey switch-on-type thinking - avoid if at all possible: What kind of object is it? Try dynamic_casts until one works - yuch!*
    - *OK: We are supposed to be able to tell the object to do this, but an error might have been made, so check the object type to be sure, and signal an error if it is the wrong type.*
  - **Be on the lookout for applications of design patterns**
    - *Many common design problems have been solved before.*
    - *Don't reinvent the wheel - apply previous patterns instead where appropriate and helpful.*
    - *Usually, multiple patterns, each with some customization, are used in a complex program.*