Week 7

Assignment 5 Review

Method Lookup

Method Lookup

- Process
  1. The variable is accessed
  2. The object stored in the variable is found
  3. The class of the object is found
  4. The class is searched for a method match
     - If no match is found, the superclass is searched
  5. This is repeated until a match is found, or the class hierarchy is exhausted

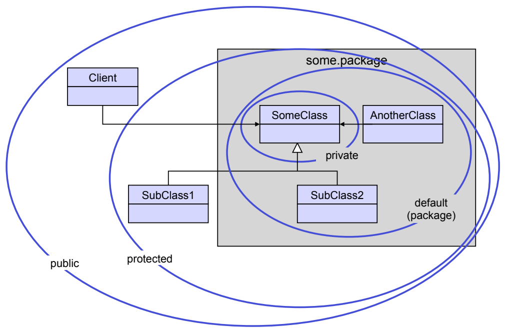
Method Lookup

- No inheritance or polymorphism
  - Method can be determined by the compiler
- Inheritance but no polymorphism
  - Inheritance hierarchy is traversed the method used is that defined in declaring class
- Inheritance and polymorphism
  - Inheritance hierarchy is traversed the first encountered method is used

Protected Access

- Private access in the superclass may be too restrictive for a subclass
- Closer inheritance relationship is supported by *protected*
- Protected access is more restricted than public access
- Keep fields private
  - Define protected accessors and mutators

Access Levels

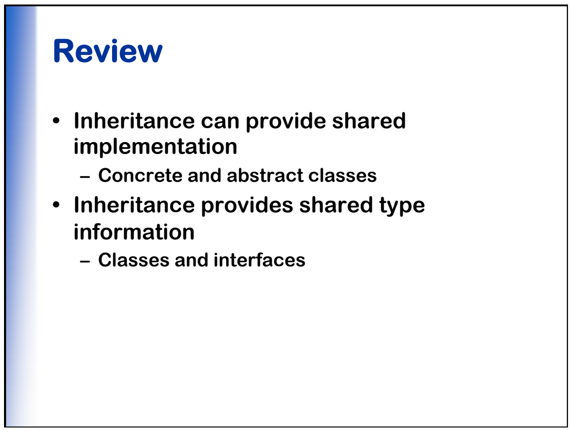


Interfaces as Types

- Implementing classes do not inherit code, but ...
- ... implementing classes are subtypes of the interface type
- So, polymorphism is available with interfaces as well as classes

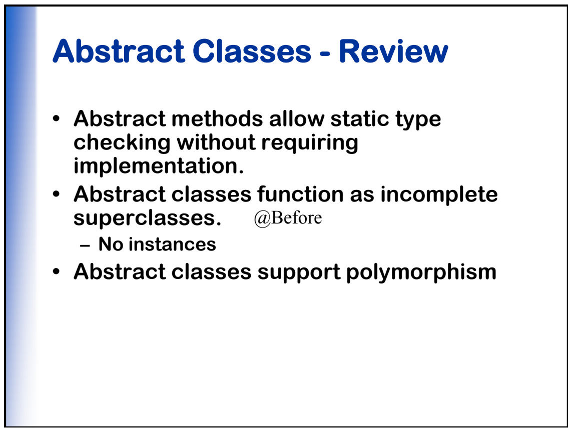
Interfaces as Specifications

- Strong separation of functionality from implementation
  - Method signature is mandated
- Clients interact independently of the implementation
  - But clients can choose from alternative implementations



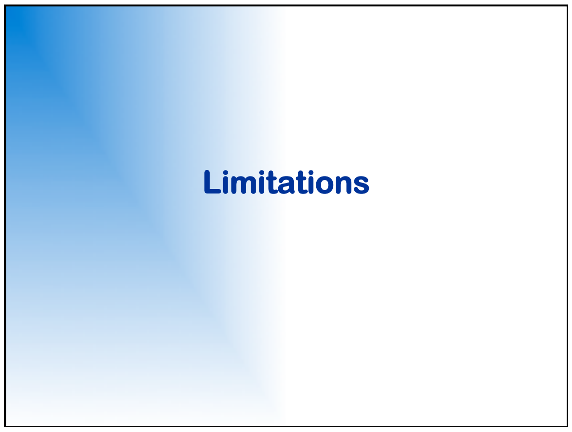
Review

- Inheritance can provide shared implementation
  - Concrete and abstract classes
- Inheritance provides shared type information
  - Classes and interfaces



Abstract Classes - Review

- Abstract methods allow static type checking without requiring implementation.
- Abstract classes function as incomplete superclasses. @Before
  - No instances
- Abstract classes support polymorphism



Limitations

Limitation to Inheritance

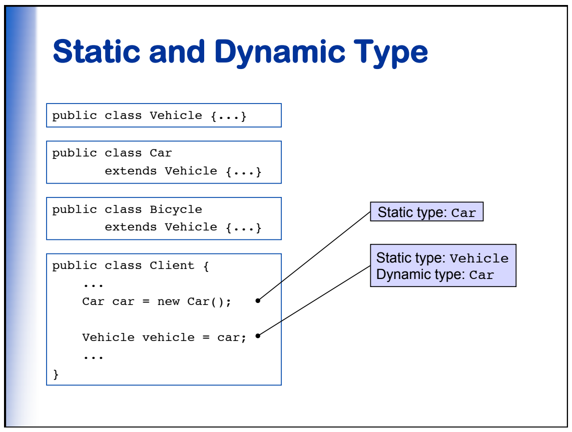
- Superclass methods can only operate on its members, not members of derived classes
- Inheritance is a one-way street:
  - A subclass inherits the superclass fields
  - The superclass knows nothing about its subclass's fields
  - Methods may be overridden by subclass
    - Implemented in every subclass?

Limitations of Subclasses

- Methods of the subclass may only operate on its members and exposed members of the superclass
  - Required members of the superclass may not be visible
  - Superclass' members may need to be `protected` visibility

Static and Dynamic Type

- The declared type of a variable is its *static type*
  - The compiler deals with static types
- The type of the object a variable refers to is its *dynamic type*
  - Used at runtime



Static and Dynamic Type

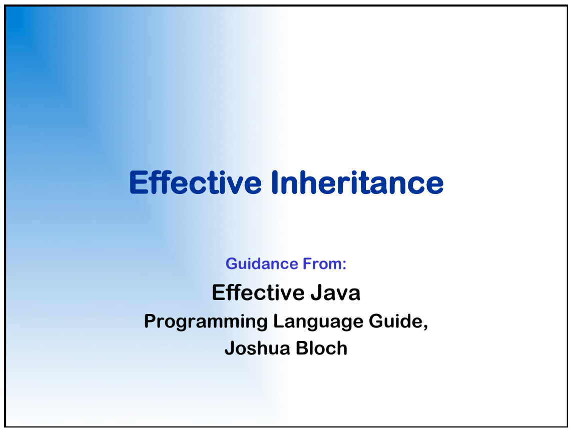
```
public class Vehicle {...}
```

```
public class Car
       extends Vehicle {...}
```

```
public class Bicycle
       extends Vehicle {...}
```

```
public class Client {
    ...
    Car car = new Car();

    Vehicle vehicle = car;
    ...
}
```



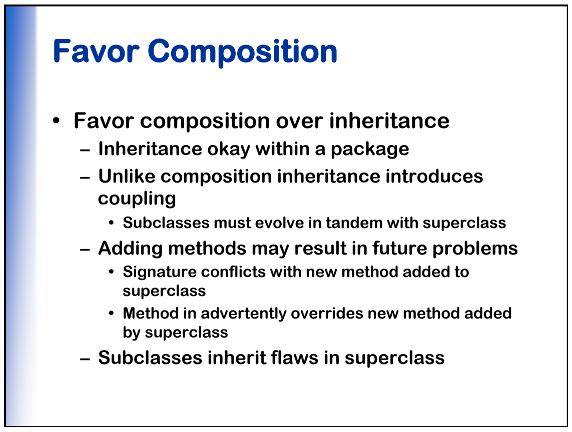
Effective Inheritance

Guidance From:

Effective Java

Programming Language Guide,

Joshua Bloch



Favor Composition

- Favor composition over inheritance
  - Inheritance okay within a package
  - Unlike composition inheritance introduces coupling
    - Subclasses must evolve in tandem with superclass
  - Adding methods may result in future problems
    - Signature conflicts with new method added to superclass
    - Method in advertently overrides new method added by superclass
  - Subclasses inherit flaws in superclass

Design for Inheritance

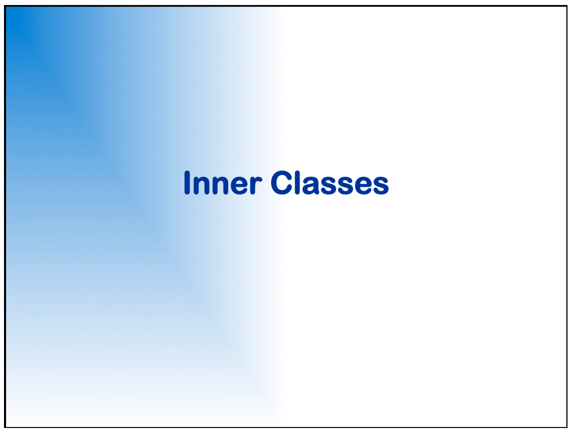
- Design and document inheritance or else prohibit it
  - Document effect of overriding each method (self-use)
    - This documents the how not the what!
  - May require exposing methods solely for the purpose of enabling extension
  - Constructors must not invoke overridable methods
  - Prohibit subclassing by
    - Making class final
    - Making constructors private or package visible (requires factory method)

Prefer Interfaces

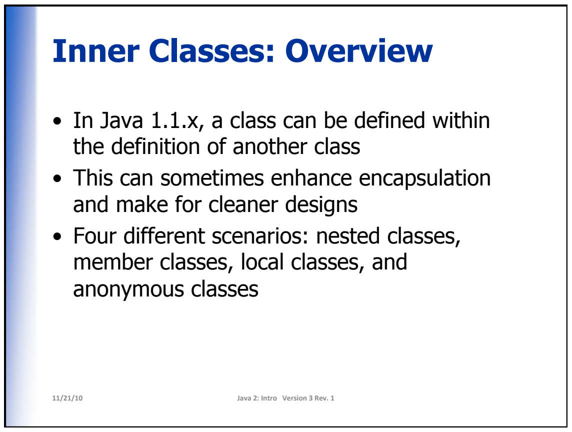
- Prefer interfaces to abstract classes
  - Existing classes may be easily retrofitted to implement an interface
  - Interfaces allow types to be defined outside rigid hierarchies
  - A skeletal implementation of the interface may ease implementation

Interfaces Define Types

- Use interfaces only to define types
  - Types should specify the kinds of things a client can do with an object
  - Interface defining a set of constants are a poor use of an interface, no type is defined
    - The static import may alleviate this temptation

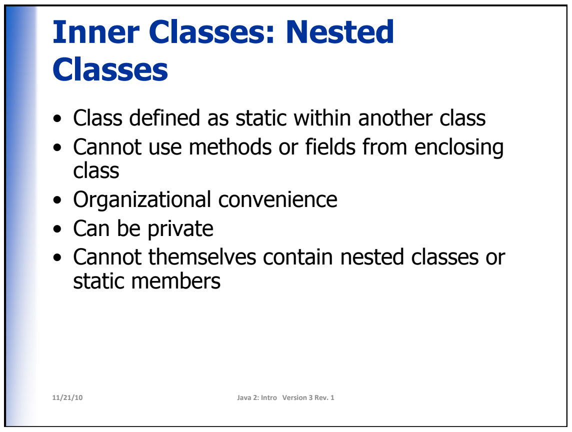


Inner Classes



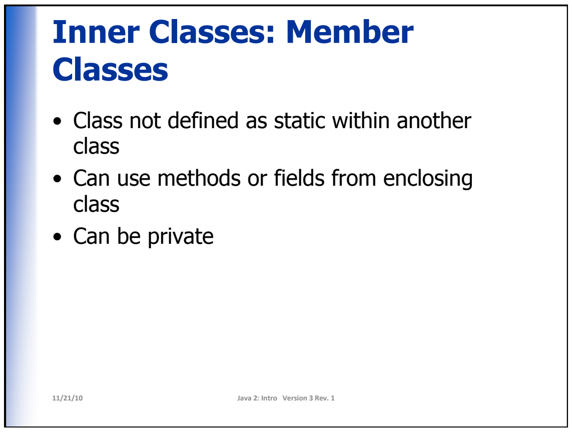
Inner Classes: Overview

- In Java 1.1.x, a class can be defined within the definition of another class
- This can sometimes enhance encapsulation and make for cleaner designs
- Four different scenarios: nested classes, member classes, local classes, and anonymous classes



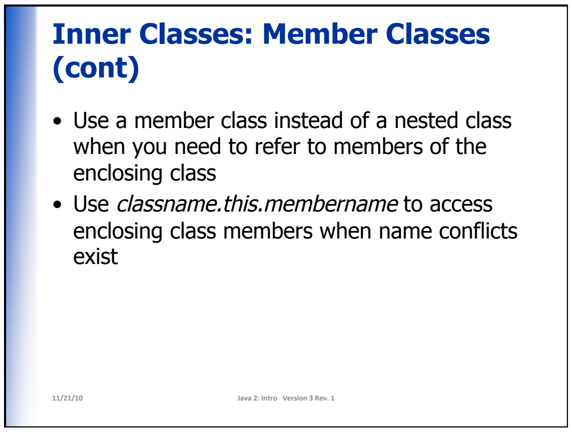
Inner Classes: Nested Classes

- Class defined as static within another class
- Cannot use methods or fields from enclosing class
- Organizational convenience
- Can be private
- Cannot themselves contain nested classes or static members



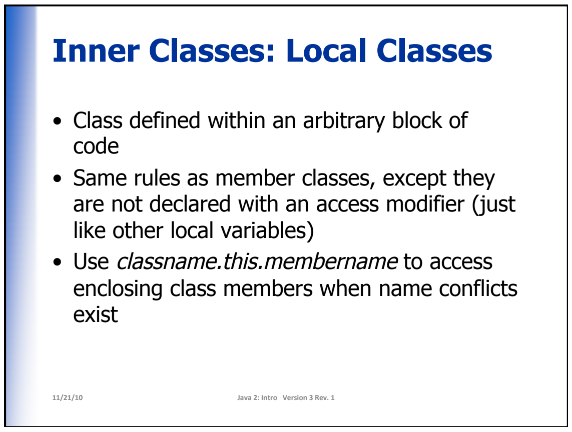
Inner Classes: Member Classes

- Class not defined as static within another class
- Can use methods or fields from enclosing class
- Can be private



Inner Classes: Member Classes (cont)

- Use a member class instead of a nested class when you need to refer to members of the enclosing class
- Use *classname.this.membername* to access enclosing class members when name conflicts exist



Inner Classes: Local Classes

- Class defined within an arbitrary block of code
- Same rules as member classes, except they are not declared with an access modifier (just like other local variables)
- Use *classname.this.membername* to access enclosing class members when name conflicts exist

Inner Classes: Anonymous Classes

- Class defined within an expression
- Exactly like a local class, except it doesn't have a name
- No constructor! (Only the compiler-supplied no-args constructor)
- Appropriate when you only need one instance of a class, and defining the class with a name doesn't clarify your code
- Class must implement a known interface or extend a known class

Inner Classes: Anonymous Classes

- Most often used in awt, where you need to create an event handler for a GUI widget
- Generally, only one instance is ever created, and is only used in that one place
- You'll use inner classes (and especially anonymous classes) next quarter