Object Oriented Programming: A Review

Based on slides from Skrien
Chapter 2

Object-Oriented Programming (OOP)

- Solution expressed as a set of communicating objects
- An object encapsulates the behavior and data related to a single concept
- Responsibility for performing a service is distributed across a set of objects

Key OOP Concepts

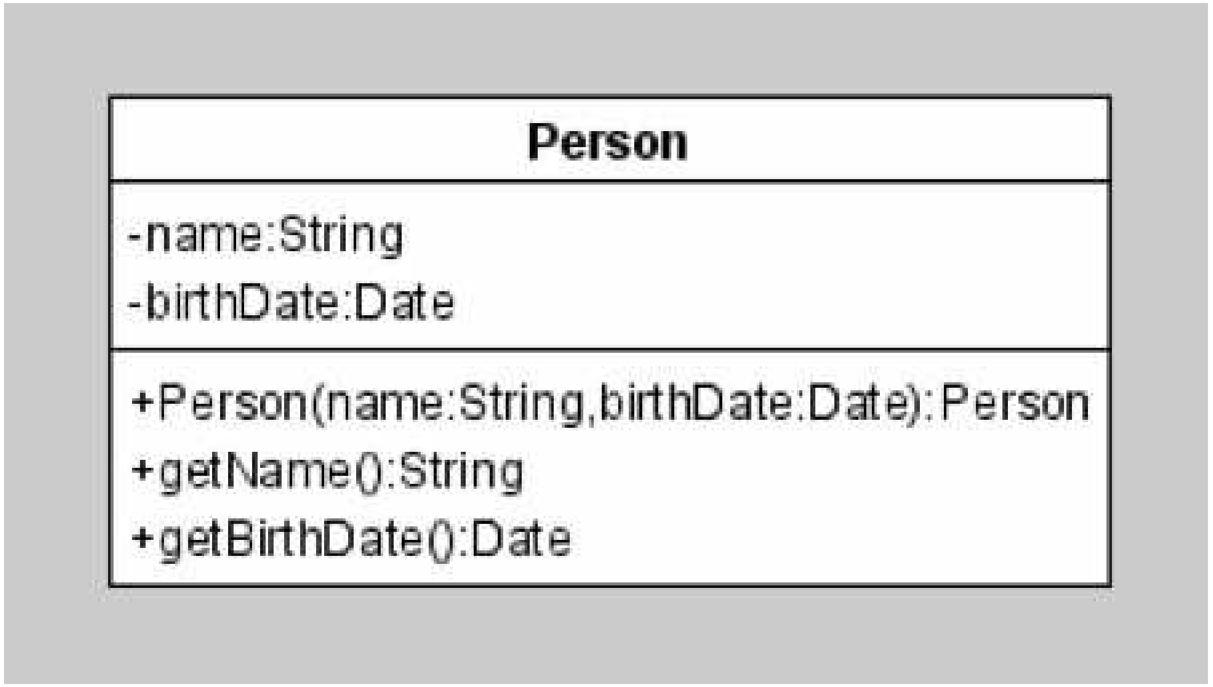
- Class: classification of objects with similar attributes and behavior
- Object: instance of a class
- Inheritance: hierarchies of classes in which classes at lower levels inherit features defined in classes at higher levels
- Message: request for an object to perform an operation on behalf of another object
- Dynamic binding: the ability to vary the object a message is sent to at run-time; the target of a message is determined at run-time, not at compile-time
- Polymorphism: the ability to substitute objects that share the same interface at run-time; dynamic binding enables polymorphism

Class

- A class describes objects that have similar features
- Two types of features
  - Data: instance variables
  - Behavior: methods

Sample Java class

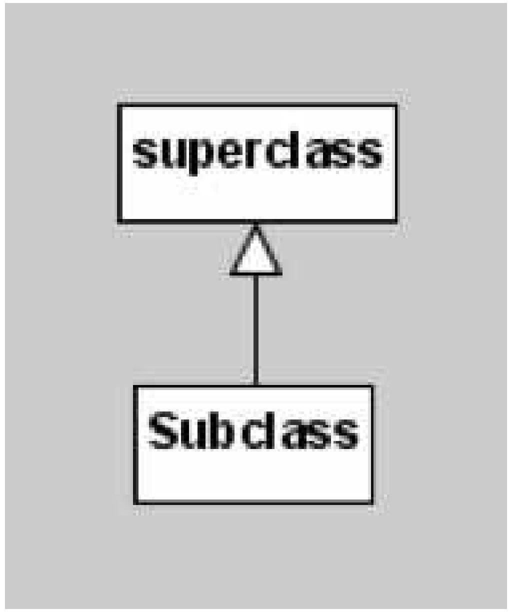
```
public class Person {
   private String name;
   private Date birthdate;

   public Person(String name, Date birthdate) {
      this.name = name;
      this.birthdate = birthdate;
   }

   public String getName() {
      return name;
   }

   public Date getBirthdate() {
      return birthdate;
   }
}
```



The UML class diagram for the Person class.

Class variables and methods

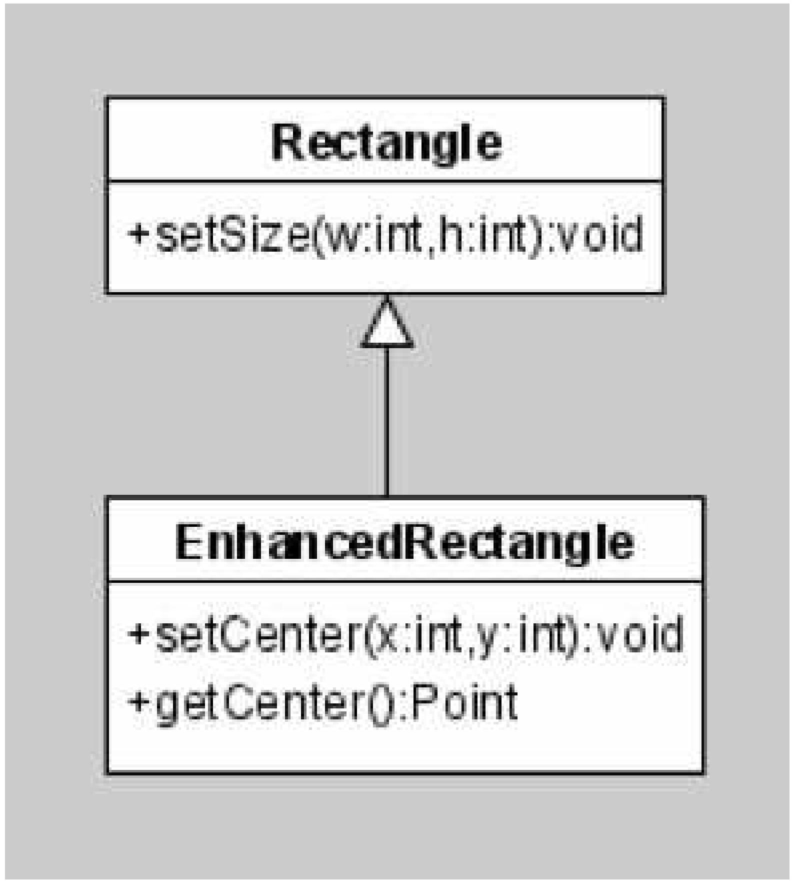
- Class (static) variables and methods should be used sparingly in OOP
- Good uses of static variables
  - To define constants
  - To define properties of a class as opposed to properties of class instances (e.g., the total number of objects in the collection)
- OK uses of static methods
  - To define methods that produce outputs using only values passed in as parameters (e.g., methods in Math class)
- See page 16 of Skrien



The UML notation for subclass and superclass.

Implementation Inheritance

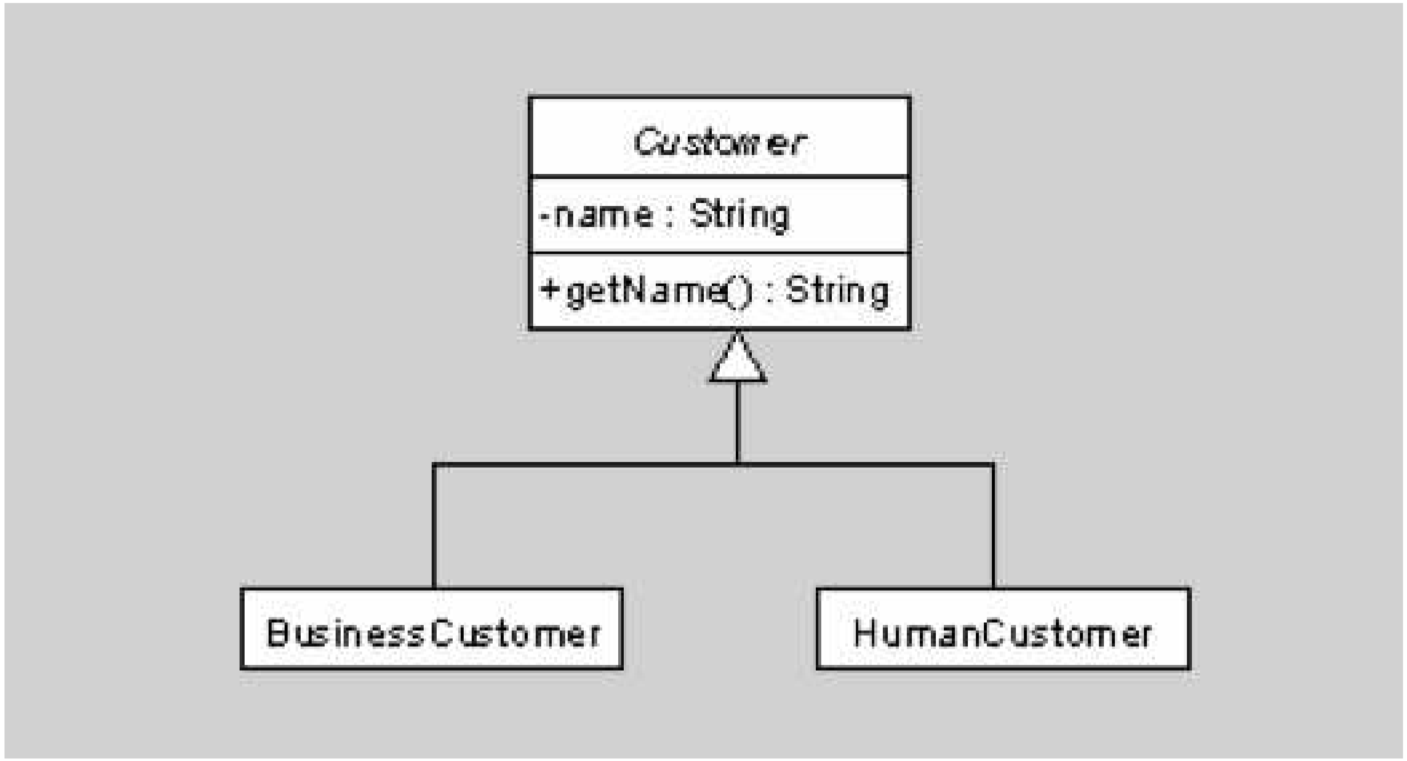
- Also called “subclassing”
- Indicated in Java by keyword “extends”
- Uses:
  - Specialize a superclass by adding new behavior in a subclass
  - Specialize a superclass by overriding existing behavior in a subclass
  - Generalize one or more subclasses by creating a superclass and moving common features up



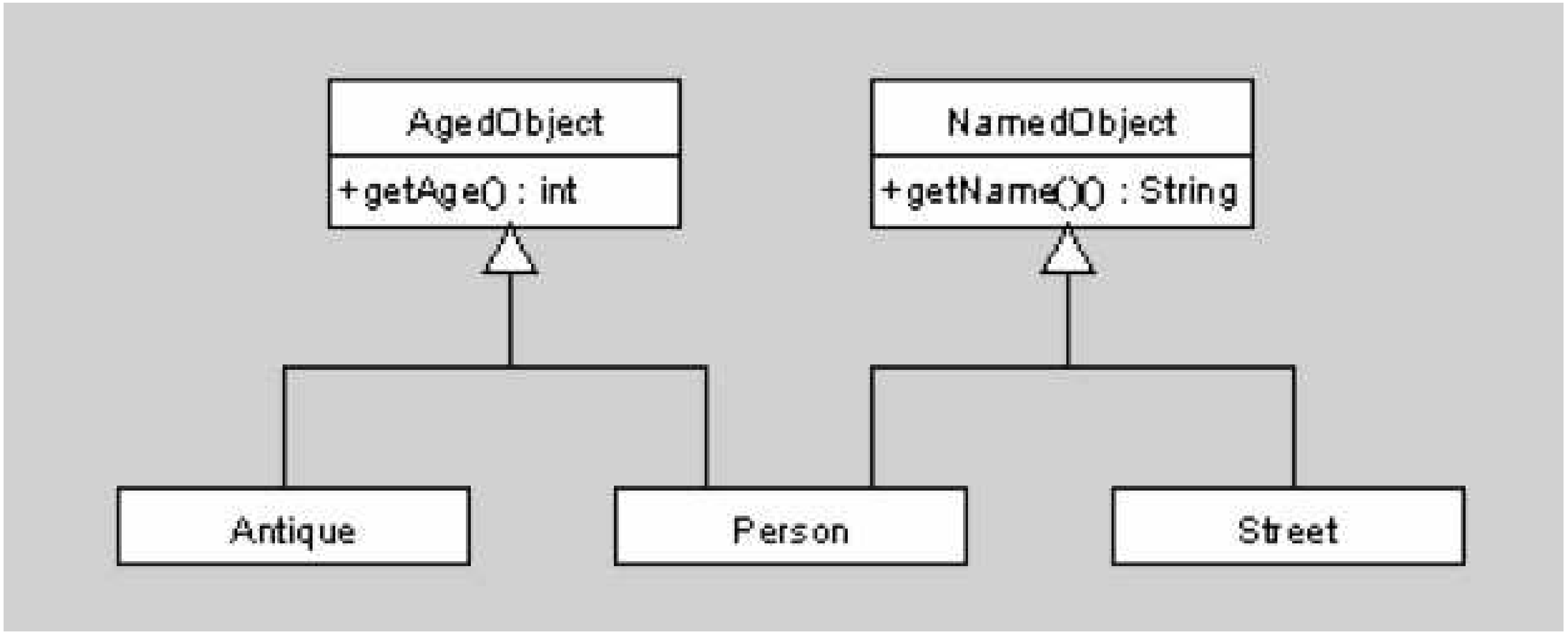
Specialization by adding new behavior to the subclass.

Specialization by overriding

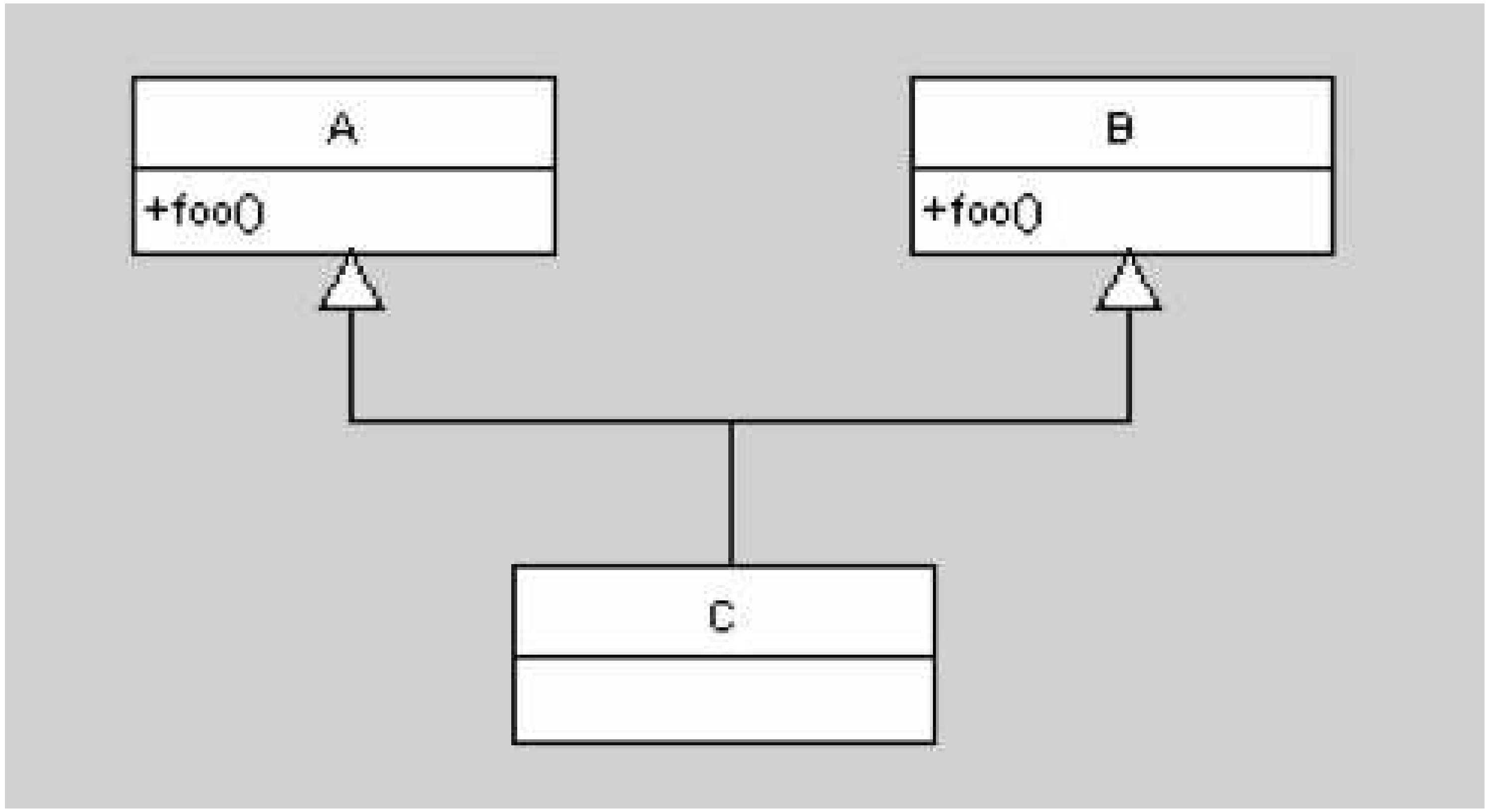
- Skrien pages 21,22: FilledOval class overrides the `draw` method of the Oval class to draw a filled-in black oval instead of a white oval with black border



Generalization by moving common behavior to a superclass.



A Person class with two superclasses. This is not legal in Java.



Class C inherits two `foo()` methods.

Types and interfaces

- A *type* is a set of data values and the operations that can be performed on them.
- Example: The Java **integer** type has 2^{32} values and has operations such as addition, subtraction, multiplication, and division.

Java classes as types

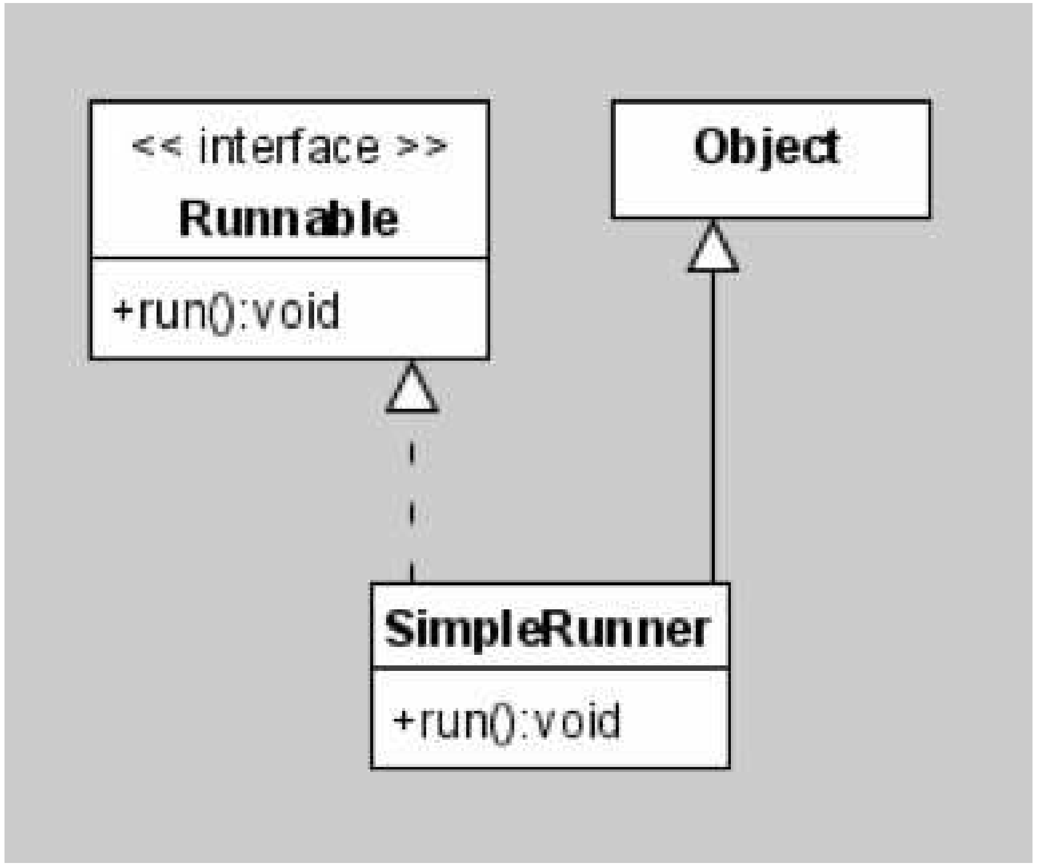
- Each Java class is a type
- The values of such a type are the objects of that class or any subclasses
- The operations are the methods of the class

Java interfaces as types

- Each Java interface is a type
- The values of such a type are the objects of all classes that implement the interface
- The operations are the methods declared in the interface

Subtypes

- A type S is a *subtype* of type T if the set of objects of S form a subset of the set of objects of T and the set of operations of S are a superset of the set of operations of T
- Example: Any subclass defines a subtype of its superclass



A SimpleRunner class that implements the Runnable interface and extends the Object class.

Subtype polymorphism

- Polymorphism: the ability to assume different forms
- You can use an object of a subtype wherever an object of a supertype is expected.
- Examples:

```
List l = new ArrayList();
List l = new LinkedList();
```

Design for change

- Suppose you used LinkedLists (LLs) in your program and now want to change to some LLs to ArrayLists (ALs) – Skrien page 27
- Change may require significant effort
  - Need to identify all places where a LL is declared or initialized and decide whether it should be changed or not
  - You may need to change calls to methods that are defined only for LLs
- Change process repeated if sometime in the future you decide to change ALs to some other type of collection

Design for change - 2

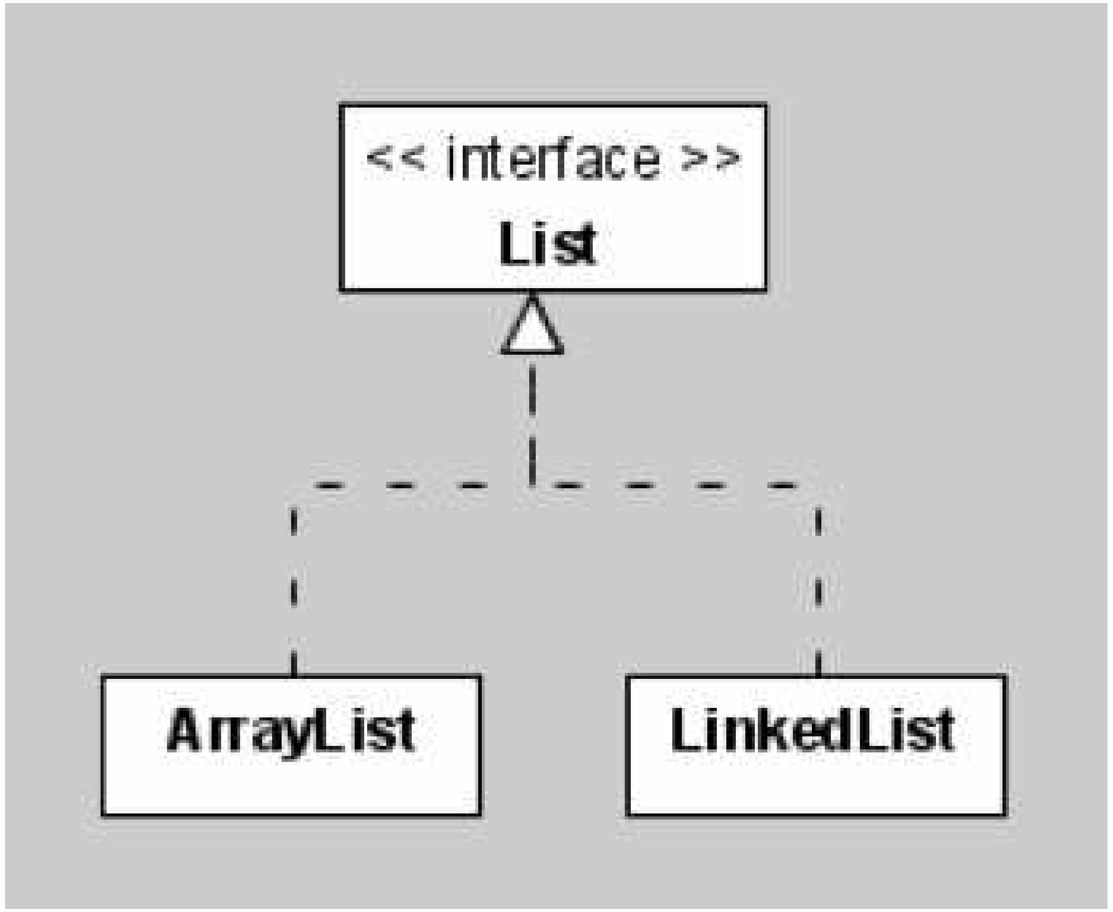
How can we design our program to accommodate change?

1. Find an appropriate interface or super class – List interface
2. Check to see if the methods provided in the super class or interface can handle all list manipulation needs
3. Declare list variables as interface or abstract class types
   - List list = new ArrayList();

A step further – factory method

- How can we minimize changes in how the variables are initialized?
  - Recall that we need to find all places where LLs are declared or initialized and decide whether to replace with ALs
- Localize code that needs to be changed
  - Factory method

```
Public class Manager {
   Public List createNewList()
   { return new LinkedList();}
}
…
List list = manager.createNewList();
```



The List interface and two classes that implement it.

Abstract classes

- Abstract classes are denoted by the key word “abstract”.
- Objects of such classes cannot be created.
- Some of the methods can be declared abstract, in which case subclasses must implement those methods (or be abstract themselves).

Abstract classes vs. interfaces

- Abstract classes can include method implementations and non-final fields
- Abstract classes with no implementations are like interfaces except a class can extend only one superclass in Java.

Dynamic method invocation

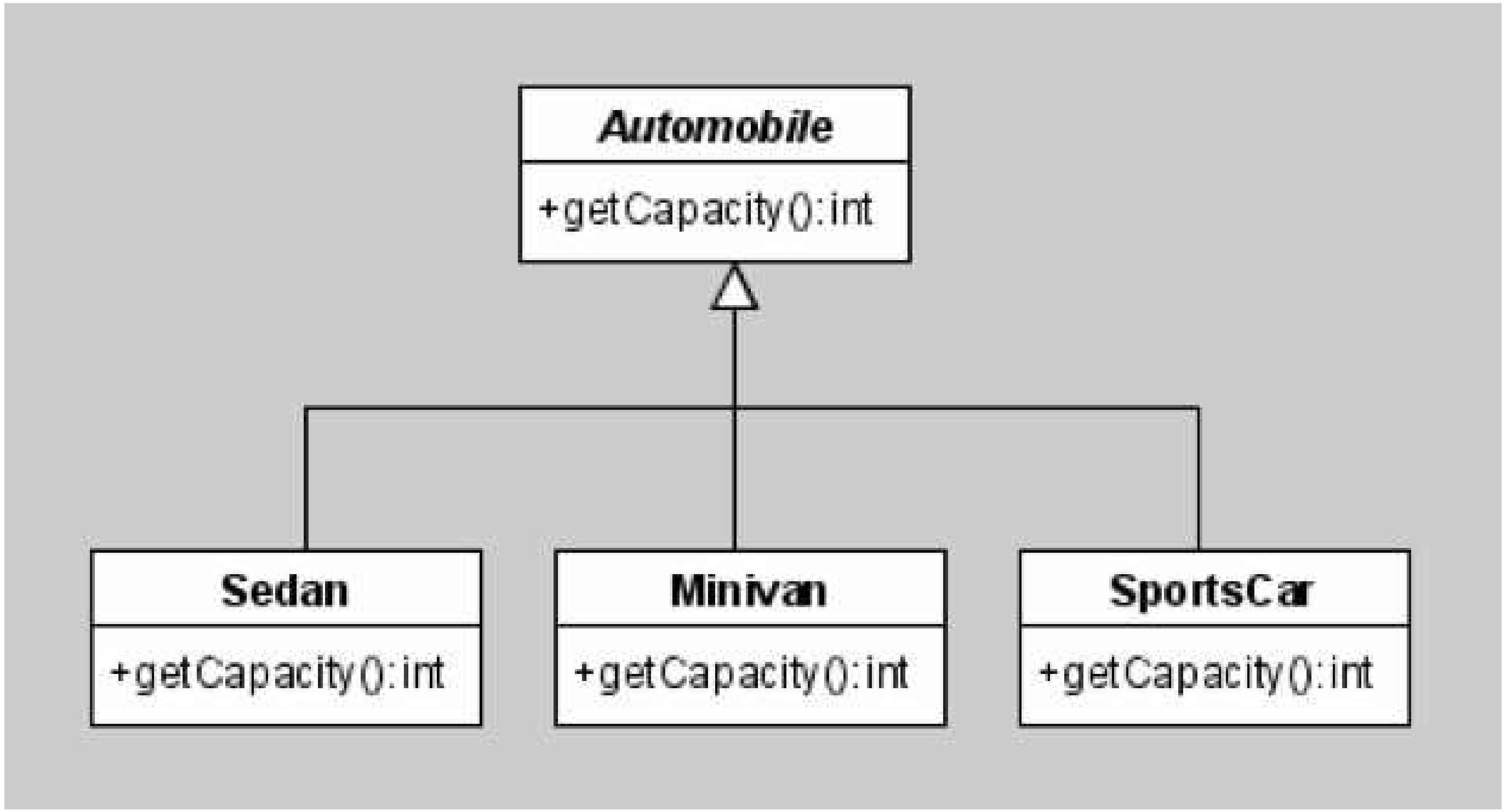
- In a method call `v.foo()`, the Java runtime environment looks at the actual class of the value of v, not the declared type of v, to determine which implementation of `foo` to execute.

Dynamic method invocation example

- Suppose a class A has a method `foo` that prints “A” and a subclass B has a method `foo` that prints “B”.

```
A v = new B();
v.foo();
```

- The implementation of `foo` in class B is the one that is executed (“B” is printed).



The abstract Automobile class and its subclasses.

Dynamic method invocation for Automobiles

```
Automobile[] fleet = new Automobile[]{
  new Sedan(Color.black),
  new Minivan(Color.blue),
  new SportsCar(Color.red)};
int totalCapacity = 0;
for(Automobile car : fleet) {
    totalCapacity += car.getCapacity();
}
```

Three different `getCapacity` methods are executed in this loop (one in each subclass)

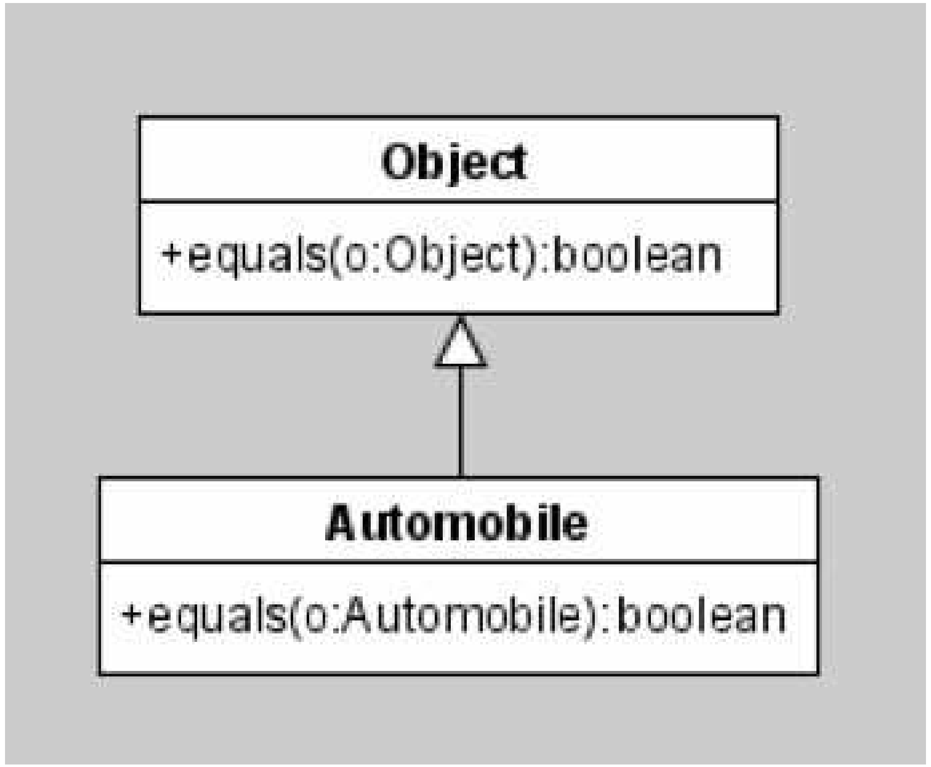
Dynamic invocation- how does it work?

```
car.getCapacity();
```

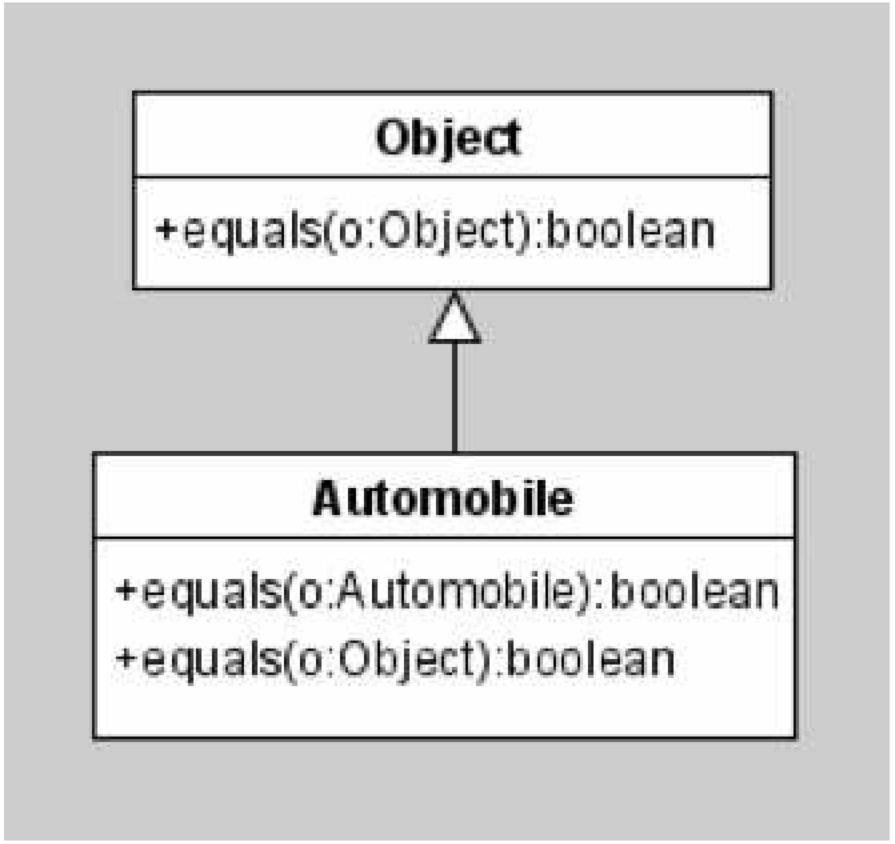
1. Compiler determines that `car` is of type `Automobile` and checks that `getCapacity()` is declared in that class
2. At runtime, the implementation (or body) of `getCapacity()` is determined based on the actual type of object stored in `car`

Overloading vs. Overriding

- *Overloading* occurs when two methods in the same class have the same name but different parameter lists.
- *Overriding* concerns two methods, one in a subclass and one in a superclass, with identical signatures (name and parameter list).



The Automobile class has two `equals` methods, one inherited and one defined in Automobile.


Object
+equals(o:Object):boolean
Automobile
+equals(o:Automobile):boolean
+equals(o:Object):boolean

The Automobile class does not inherit the Object class' `equals` method. Instead, it overrides that method.

Overloaded method example

```
Object o = new Object();
Automobile auto = new Automobile();
Object autoObject = new Automobile();
auto.equals(o);
auto.equals(auto);
auto.equals(autoObject);
```

Which of the two `equals` methods in the Automobile class is executed in each of the last 3 lines above?