Algorithms

1. Objects and Classes

Robert O’Connor, Frank Walsh

Objectives

- Objects and Classes
- Using Methods in a Java Class
  - References and Aliases
  - Arguments and Parameters
- Defining a Java Class
  - Passing Arguments
  - Constructors
  - The **toString** Method
  - Static Fields and Methods
- Packages – The Java Class Library

Objectives

- Composition
  - Adapters
- Inheritance
  - Invoking constructors from within constructors
  - Private fields and methods of the base class
  - Overriding, overloading methods
  - Protected access
  - Multiple inheritance
  - Type compatibility and base classes
  - The class **Object**
  - Abstract classes and methods
- Polymorphism

Objectives

- Encapsulation
- Specifying Methods
- Java Interfaces
  - Writing an Interface
  - Implementing an Interface
  - An Interface as a Data Type
  - Type Casts Within an Interface Implementation
  - Extending an Interface
  - Named Constants Within an Interface
  - Interfaces Versus Abstract Classes

Objects

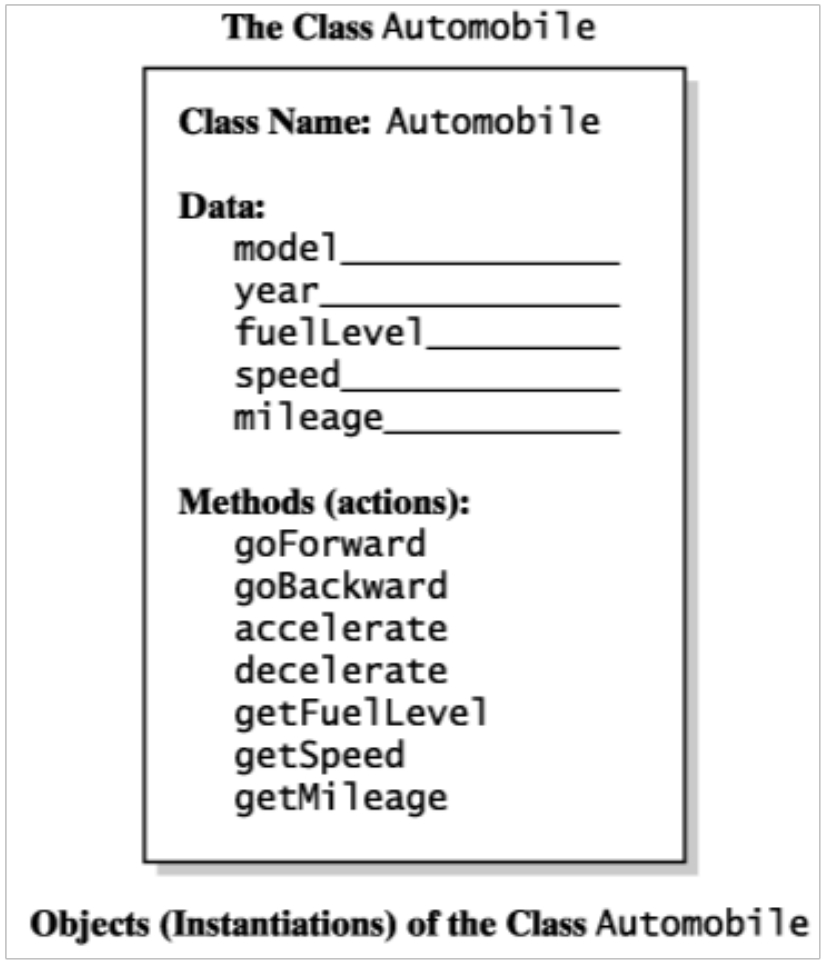
- An object is a program construct that
  - Contains data
  - Performs certain actions
- The actions are called methods
- The actions interact to form the solution to a given problem

Classes

- A class is a type or kind of object
- Objects of the same class have
  - The same kinds of data
  - The same methods
- A class definition is a general description of
  - What the object is
  - What it can do

Classes

Fig. 1-1 An outline of a class



Class Instantiation

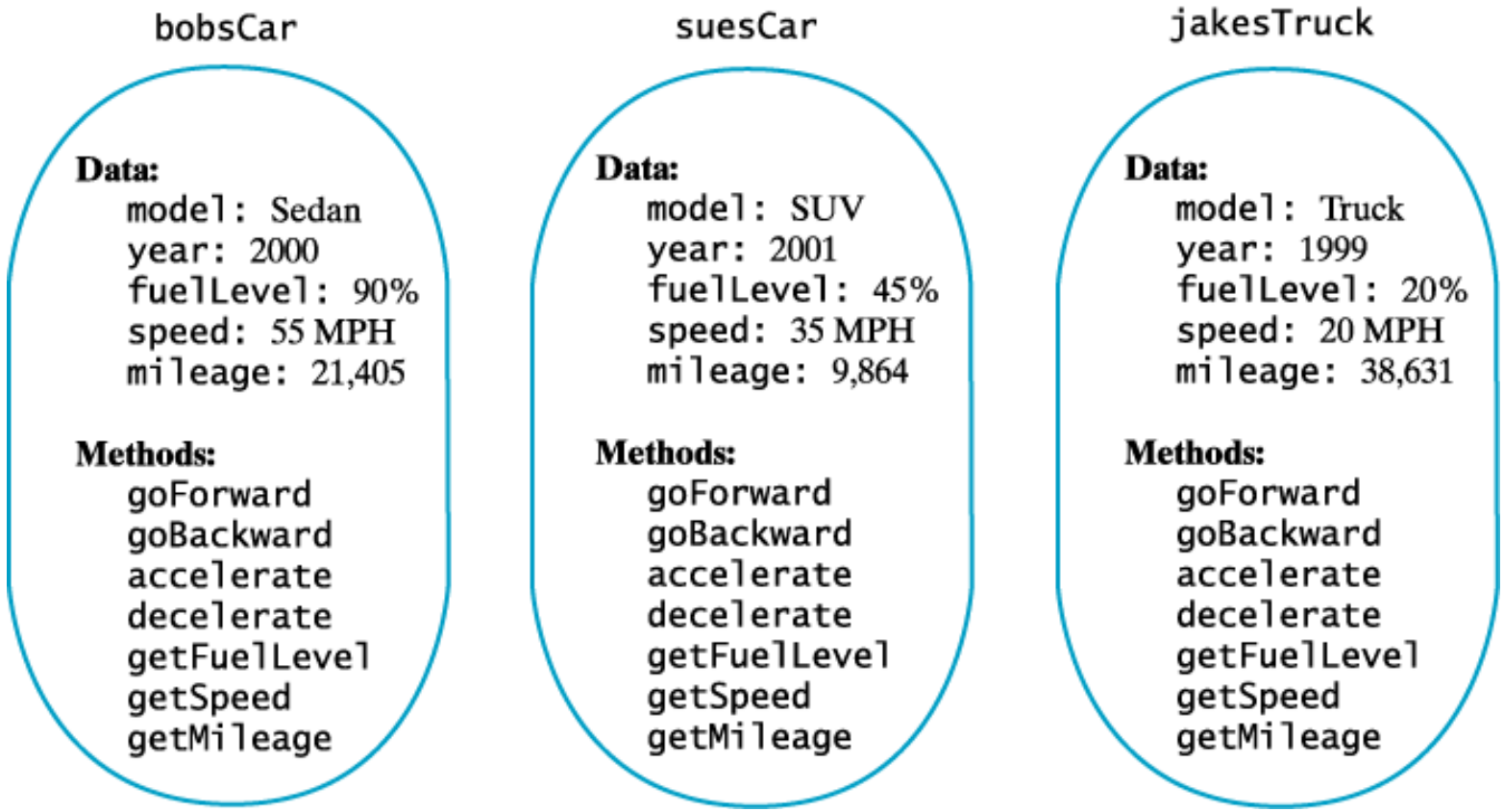


Fig. 1-2 Three instances of the class automobile

Methods in Java

- Given: **Name joe = new Name();**
- The **new** operator creates an instance of the class
  - Invokes the constructor method
- Valued methods return a single value
- **void** methods do not return a value

Methods in a Java Class

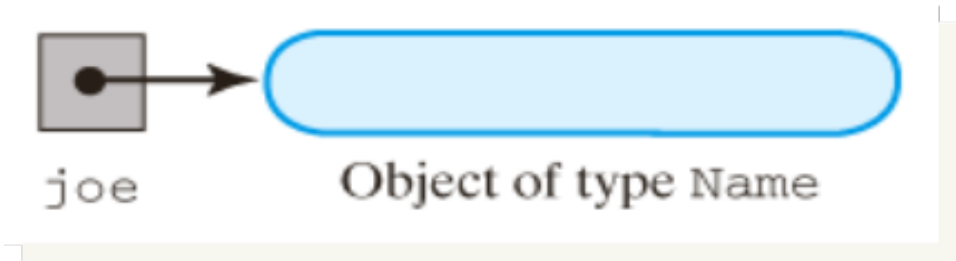


Fig. 1-3 A variable that references an object.

References and Aliases

- Primitive types:
  byte, short, int, long
  float, double, char, boolean
- All other types are reference or class types
  String greeting = "Howdy";
  - **greeting** is a reference variable
- When two variables reference the same instance, they are considered aliases

References and Aliases

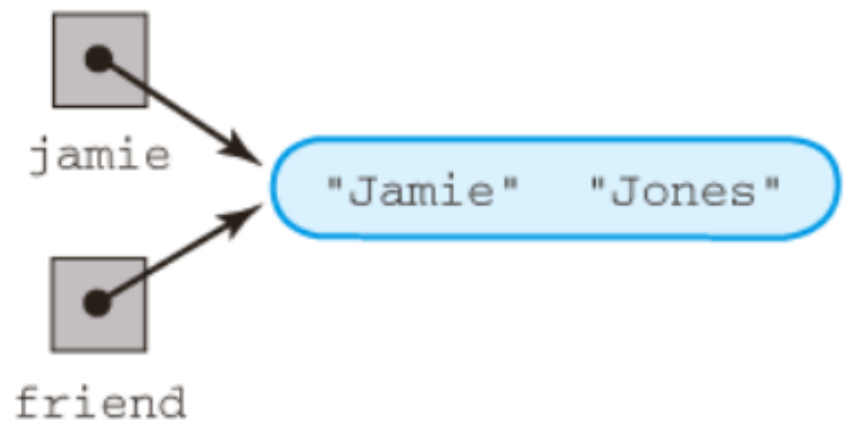


Fig. 1-4 Aliases of an object

Arguments and Parameters

- Given

  Name joe = new Name();
  joe.setFirst ("Joseph");
  joe.setLast ("Brown");

- **"Joseph"** and **"Brown"** are arguments sent to the methods
- Invocation of method must have same number of arguments as there are formal parameters in the declaration

Defining a Class

- Given

```
public class Name
{ private String first; // first name
private String last; // last name
< Definitions of methods are here. >
} // end Name
```

- These are the data fields (instance variables)
- Note they are private
  - They will require accessor and mutator methods

Methods

- Given

```
public String getLast()
{ return last; } // end getLast
```

  - This is a valued method
  - Returns a **String**

- Given

```
public void setLast(String lastName)
{ last = lastName; } // end setLast
```

- This is a void method

Naming Convention

- Start method name with lowercase letter
  - Use verb or action phrase **e.g. getLast**
- Start class name with uppercase
  - Use noun or descriptive phrase

  e.g public class Name
- Local variables
  - A variable declared within a method

Passing Arguments

- Call by value
  - For primitive type, parameter initialised to value of argument in call
- Call by reference
  - For a class type, formal parameter is initialised to the address of the object in the call

```
public void giveLastNameTo(Name child)
{ child.setLast(last);
} // end giveLastNameTo
```

Passing Arguments

Name jamie = New Name(“Jamie”, “Jones”);

Name jane = New Name(“Jane”, “Doe”);

jamie.giveLastNameTo(jane);

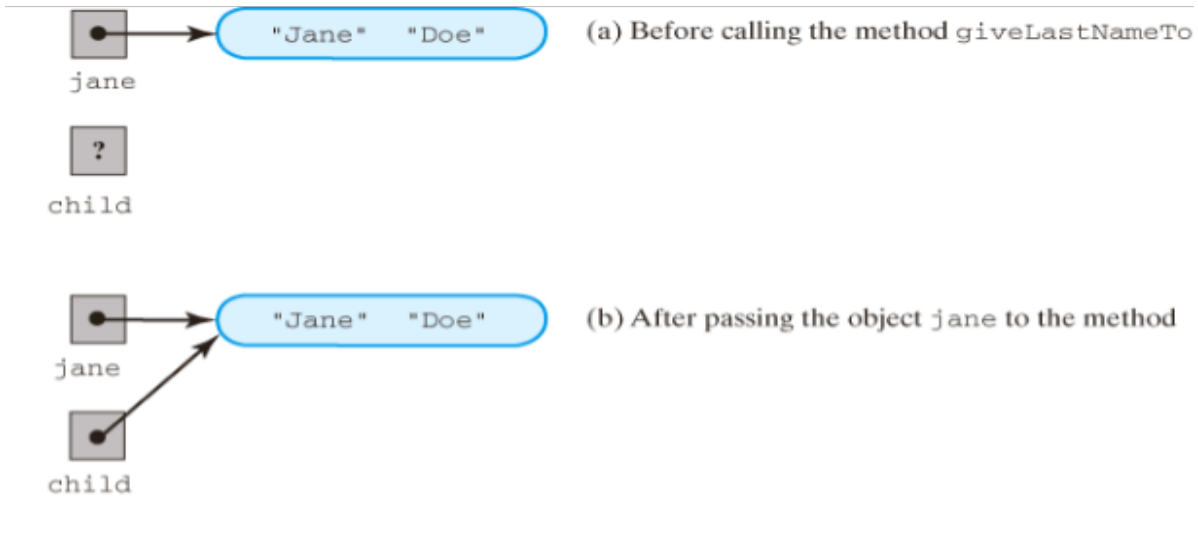


Fig.1-5 a & b The method **giveLastNameTo** modifies the object passed to it as an argument.

Also see:

http://www.javaworld.com/javaworld/javaqa/

Constructors

- A method that
  - Allocates memory for the object
  - Initialises the data fields
- Properties
  - Same name as the class
  - No return type, not even void
  - Any number of formal parameters
- Often method name = class name

Constructors

```
public Name (String firstName, String lastName) {
    first = firstName;
    last = lastName;
} // end constructor
.....
Name jill = new Name("Jill", "Jones");
```

(a)

Name jill = new Name ("Jill", "Smith");

Fig. 1-7 An object (a) after its initial creation; (b) after its reference is lost

Static Fields & Methods

- A data field that does not belong to any one object
- One instance of that data item exists to be shared by all the instances of the class
- Also called:
  static field, static variable, class variable

Static Fields & Methods

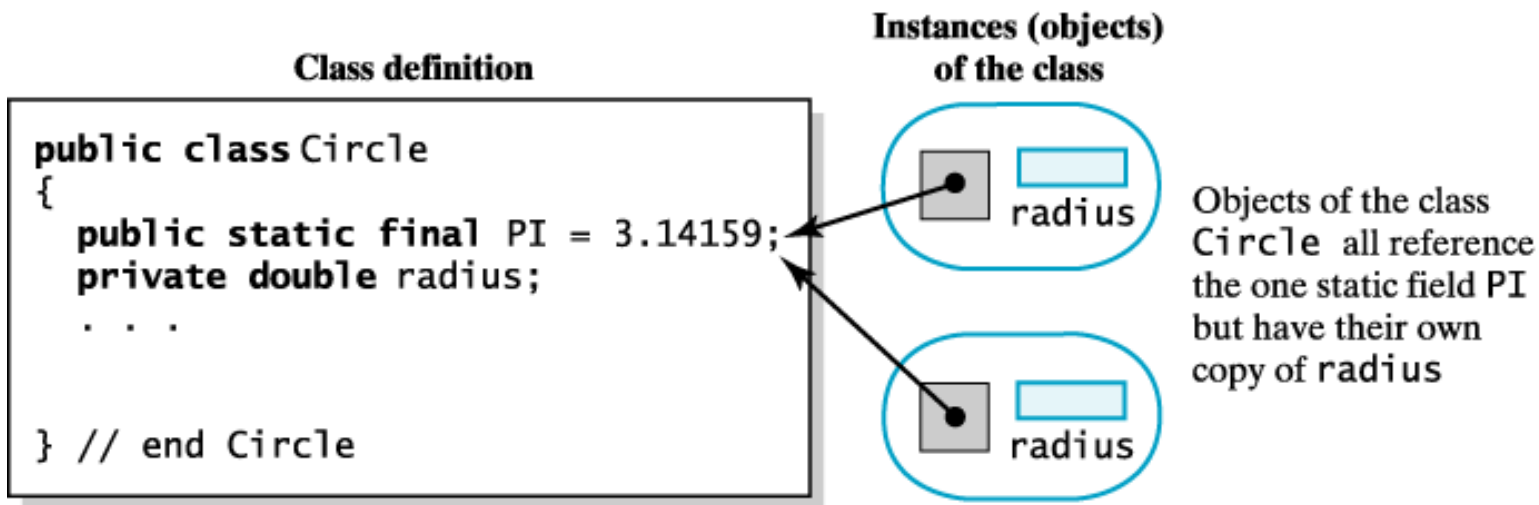


Fig. 1-8 A static **PI** versus a non static field

Packages

- Multiple related classes can be conveniently grouped into a package
- Begin each file that contains a class within the package

  package myStuff;
- Place all files within a directory
  - Give folder same name as the package
- To use the package, begin the program with

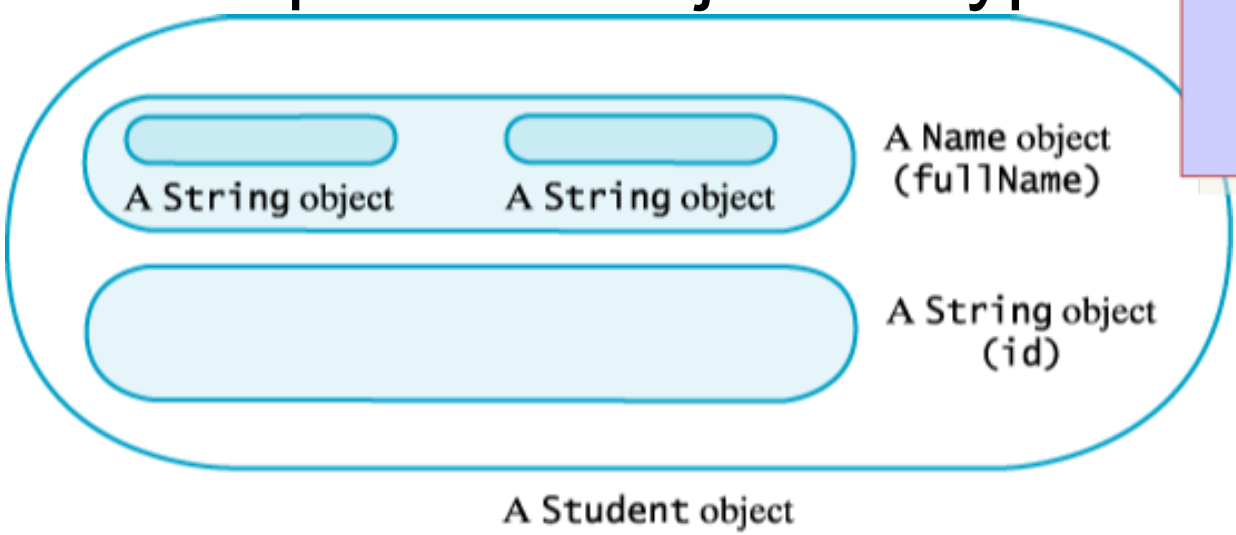
  import myStuff.*;

Java Class Libraries

- Many useful classes have already been declared
- Collection exists in Java Class Library
- Example
  - The class **Math** is part of the package **java.lang**

Composition

- When a class has a data field that is an instance of another class
- Example – an object of type **Student**



A "has a" relationship

Fig. 1-9 A **Student** object composed of other objects

Adapters

Use composition to write a new class:

- Has an instance of an existing class as a data field
- Defines new methods needed for the new class

Example:

```
public class NickName
{ private Name nick;
public NickName()
{ nick = new Name(); }// end default constructor
public void setNickName(String nickName)
{ nick.setFirst(nickName); }// end setNickName
public String getNickName()
{ return nick.getFirst(); }// end getNickName
} // end NickName
```

Inheritance

- A general or base class is first defined
- Then a more specialised class is defined by …
  - Adding to details of the base class
  - Revising details of the more general class
- Advantages
  - Saves work
  - Common properties and behaviors are define only once for all classes involved

Inheritance

```
public class Car extends Automobile
{ ...
} // end Car
…
Car toyota = new Car();
```

An "is a" relationship

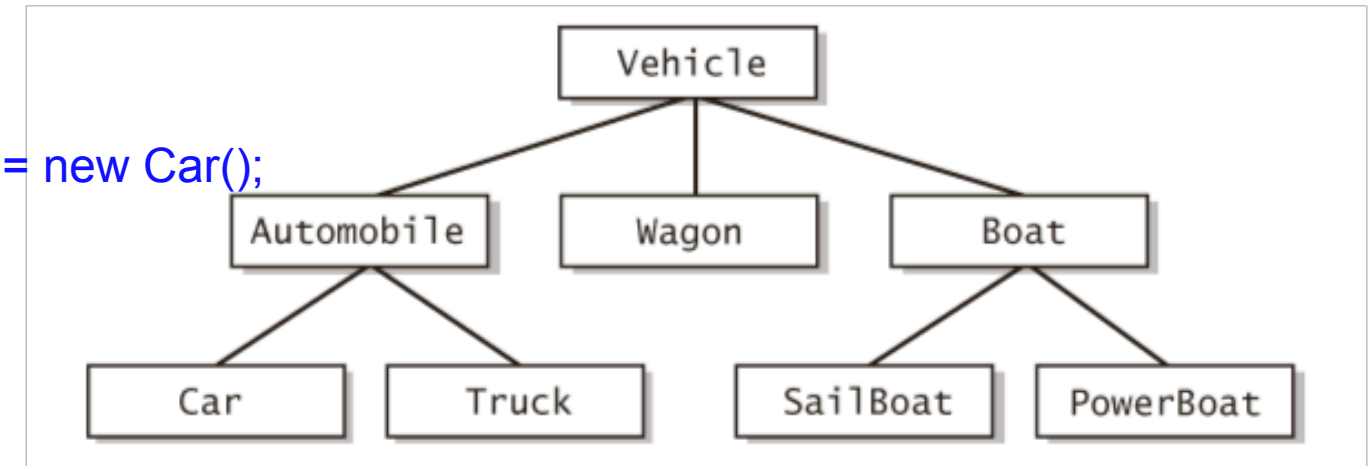


Fig. 1-10 A hierarchy of classes.

Base Class Constructor

- Constructors usually initialise data fields
- In a derived class
  - The constructor must call the base class constructor
- Can use the reserved word **super** as a name for the constructor of the base class
  - When **super** is used, it must be the first action in the derived constructor definition
  - Must not use the name of the constructor

Base Class Constructor

```
public class Automobile extends Vehicle{
private int year; // year of manufacture
private String colour;
public Automobile(int iYear, String sColour){
year = iYear;
colour = sColour;
}
} // end Automobile
```

```
public class Car extends Automobile{
private int engineSize;
private String modelType; // saloon, hatchback

public Car (int iYear, String sColour, int iEngine,
String sModel){
super (iYear, sColour);
engineSize = iEngine;
modelType = sModel;
}
} // end Car
```

Accessing Inherited Data Fields

- Private data field in base class
  - Not accessible by name within definition of a method from another class – including a derived class
  - Still they are inherited by the derived class
- Derived classes must use public methods of the base class **e.g. getModel();**
- Note that private methods in a base class are also unavailable to derived classes
  - But usually not a problem – private methods are used only for utility duties within their class

Overriding Methods

- When a derived class defines a method with the same signature as in base class
  - Same name
  - Same return type
  - Same number, types of parameters
- Objects of the derived class that invoke the method will use the definition from the derived class
- It is possible to use **super** in the derived class to call an overridden method of the base class

Overriding Methods

```
public class Automobile extends Vehicle{
private int year; // year of manufacture
private String colour;
…
public String toString(){
return year + “, “ + colour;
}
} // end Automobile
```

```
public class Car extends Automobile{
private int engineSize;
private String modelType; // saloon, hatchback
….
public String toString(){
return super.toString() + “, “ + engineSize + “, “ +
modelType;
}
} // end Car
```

Overriding Methods

- Multiple use of **super**
  - Consider a class derived from a base … that itself is derived from a base class
  - All three classes have a method with the same signature
- The overriding method in the lowest derived class cannot invoke the method in the base class's base class
  - The construct **super.super** is illegal

Overloading Methods

- When the derived class method has
  - The same name
  - The same return type … but …
  - Different number or type of parameters
- Then the derived class has available
  - The derived class method … and
  - The base class method with the same name
- Java distinguishes between the two methods due to the different parameters

Overloading Methods

- A programmer may wish to specify that a method definition cannot be overridden
  - So that the behavior of the constructor will not be changed
- This is accomplished by use of the modifier **final**

```
public final void whatever()
{
. . .
}
```

Protected Access

- A method or data field modified by **protected** can be accessed by name only within
  - Its own class definition
  - Any class derived from that base class
  - Any class within the same package

Protected Access

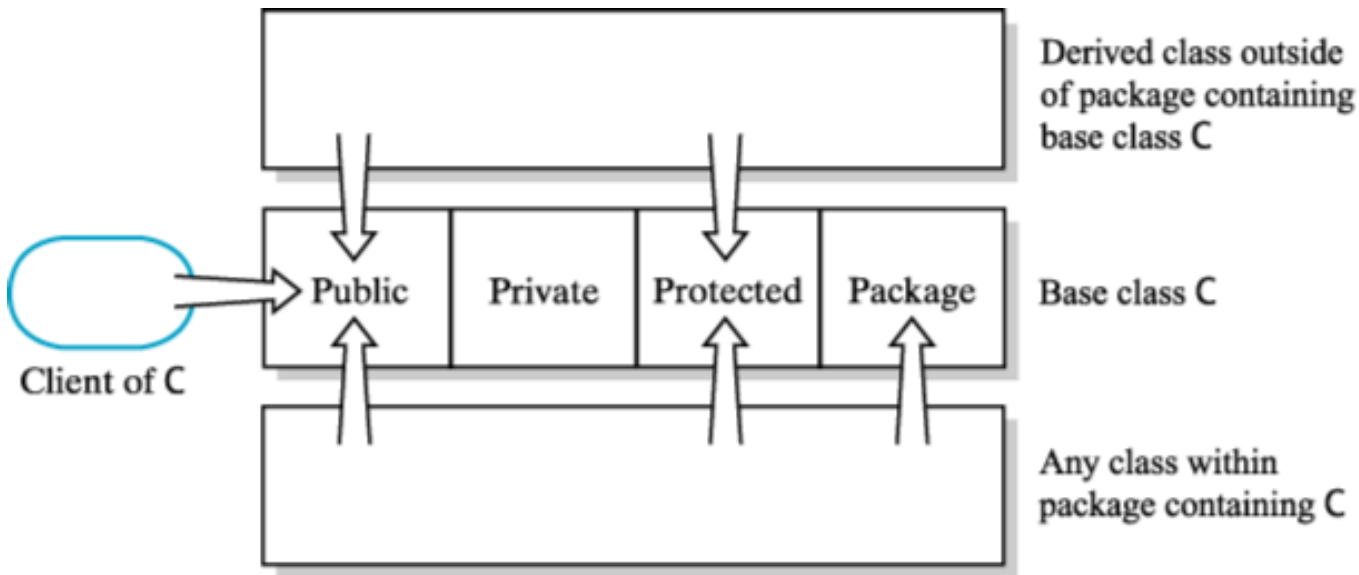


Fig. 1-11 Public, private, protected, and package access

Multiple Inheritence

- Some languages allow programmer to derive class **C** from classes **A** and **B**
- Java does not allow this
  - A derived class can have only one base class
- Multiple inheritance can be approximated
  - A derived class can have multiple interfaces

Object Types of Derived Classes

- Given :
  - Class **Car**,
  - Derived from class **Automobile**
- Then a **Car** object is also an **Automobile** object
- In general …
  An object of a derived class is also an object of the base class

The class Object

- Every class is a descendant of the class **Object**
- **Object** is the class that is the beginning of every chain of derived classes
  - It is the ancestor of every other class
  - Even those defined by the programmer

Abstract Classes

- Some base classes are not intended to have objects of that type
  - The objects will be of the derived classes
- Declare that base class to be abstract

```
public abstract class Whatever
{ . . . }
```

- The designer often specifies methods of the abstract class without a body

```
public abstract void doSomething();
```

  - This requires all derived classes to implement this method

Polymorphism

- When one method name in an instruction can cause different actions
  - Happens according to the kinds of objects that invoke the methods
- Example

```
public void displayAt(int numLines){
for (int count=0;count<numLines;count++){
system.out.println();}
display();
}
```

```
UndergradStudent ug = new UndergradStudent(. . .);
Student s = ug; // s and ug are aliases
s.displayAt(2);
ug.displayAt(4);
```

The object still remembers it is of type **UndergradStudent**

Polymorphism

- Which **displayAt** is called …
  - Depends on the invoking object's place in the inheritance chain and is not determined by the type of the variable naming the object

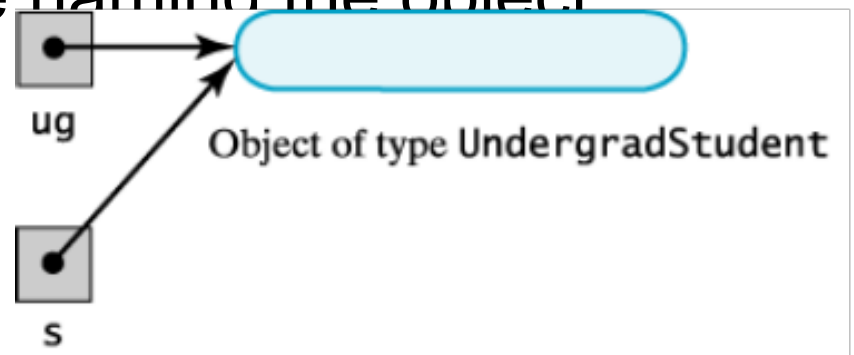


Fig. 1-12 The variable “s” is another name for an undergraduate object.

http://download.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Encapsulation

- Hides the fine detail of the inner workings of the class
  - The implementation is hidden
  - Often called "information hiding"
- Part of the class is visible
  - The necessary controls for the class are left visible
  - The class interface is made visible
  - The programmer is given only enough information to use the class

Encapsulation

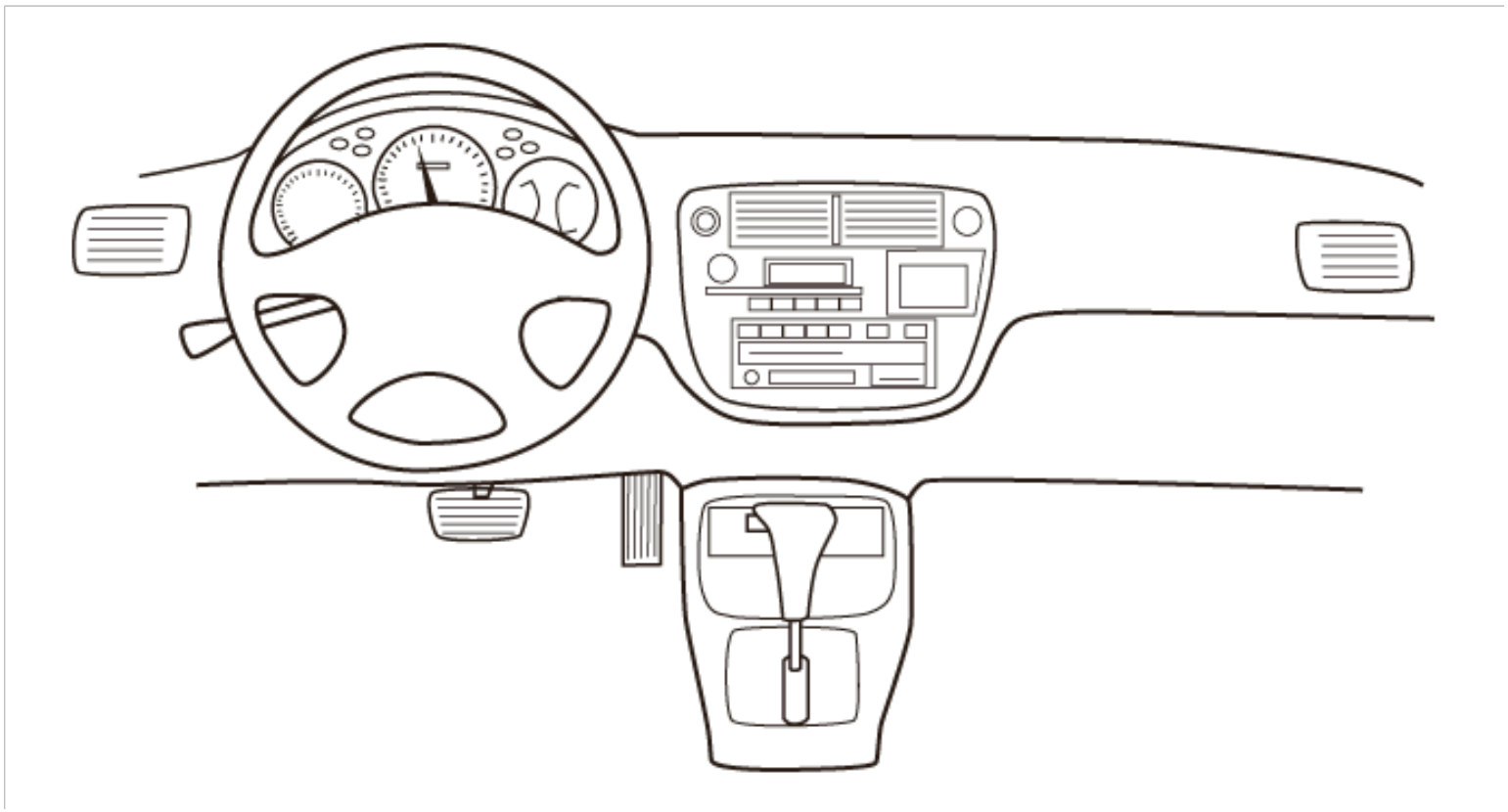


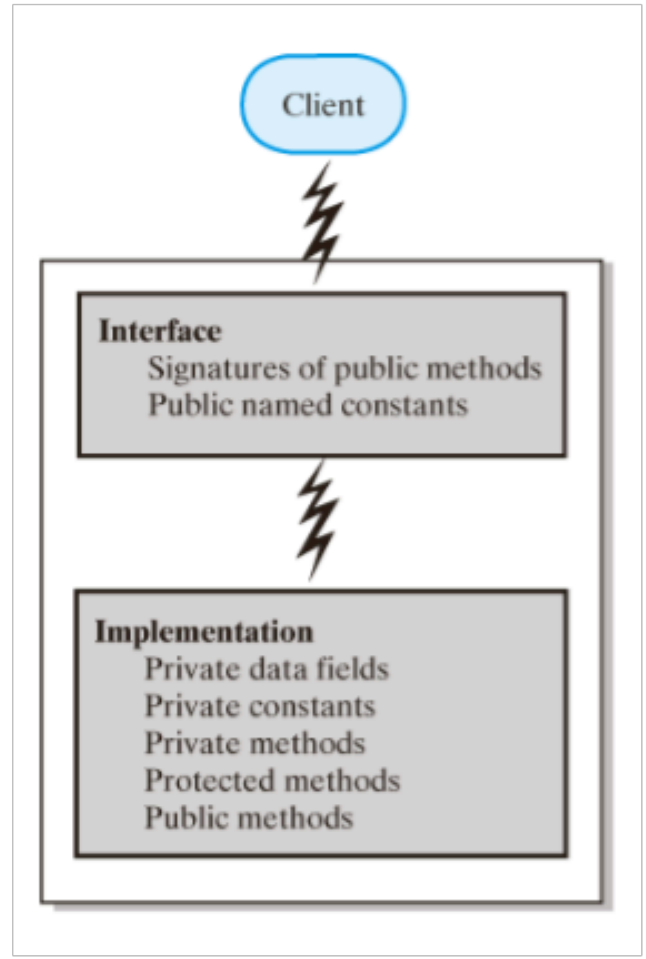
Fig. 1-13 An automobile's controls are visible to the driver, but its inner workings are hidden.

Abstraction

- A process that has the designer ask what instead of how
  - What is it you want to do *with* the data
  - What will be done *to* the data
- The designer does not consider how the class's methods will accomplish their goals
- The client interface is the what
- The implementation is the how

Abstraction

Fig. 1-14 An interface provides well-regulated communication between a hidden implementation and a client.



Specifying Methods

- Specify what each method does
- Precondition
  - Defines responsibility of client code
- Postcondition
  - Specifies what will happen if the preconditions are met
- Assertions can be written as comments to identify design logic

```
// Assertion: intVal >= 0
```

Java Interface

- A program component that contains
  - Public constants
  - Signatures for public methods
  - Comments that describe them
- Begins like a class definition
  - Use the word **interface** instead of **class**

```
public interface someClass
{
public int someMethod();
}
```

Java Interface Example

```
public interface NameInterface
{ /** Task: Sets the first and last names.
* @param firstName a string that is the desired first name
* @param lastName a string that is the desired last name */
public void setName(String firstName, String lastName);
/** Task: Gets the full name.
* @return a string containing the first and last names */
public String getName();
public void setFirst(String firstName);
public String getFirst();
public void setLast(String lastName);
public String getLast();
public void giveLastNameTo(NameInterface child);
public String toString();
} // end NameInterface
```

Implementing an Interface

- A class that implements an interface must state so at start of definition

  public class myClass implements someInterface

- The class must implement every method declared in the interface
- Multiple classes can implement the same interface
- A class can implement more than one interface
- An interface can be used as a data type

  public void someMethod (someInterface x)

Implementing an Interface

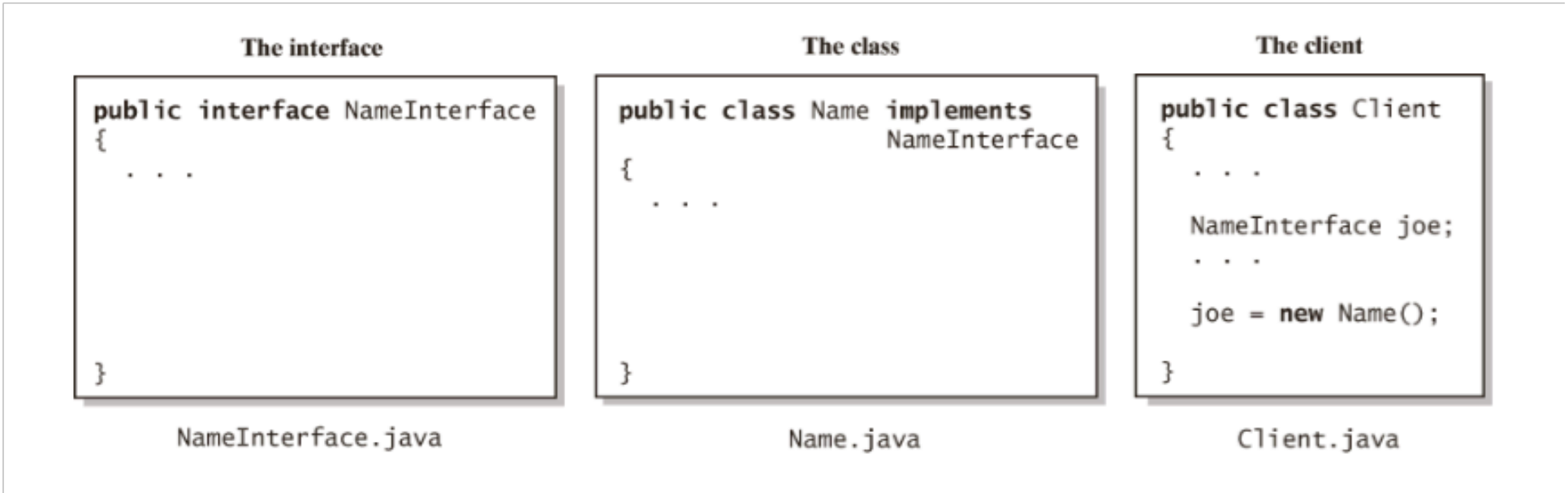


Fig. 1-15 The files for an interface, a class that implements the interface, and the client.

Extending an Interface

- Use inheritance to derive an interface from another
- When an interface extends another
  - It has all the methods of the inherited interface
  - Also include some new methods
- Also possible to combine several interfaces into a new interface
  - Not possible with classes

Extending an Interface

```
public interface Nameable{
public void setName(String petName);
public String getName();
} // end Nameable
…
public interface Callable extends
Nameable{
public void come(String petName);
} // end Callable
…
public interface Capable{
public void hear();
public void respond();
} // end capable
```

```
public interface Trainable extends Callable,
Capable{
public void sit();
public void speak();
public void lieDown();
} // end Trainable
```

Named Constants

- An interface can contain named constants
  - Public data fields initialised and declared as **final**
- Consider an interface with a collection of named constants
  - Then derive variety of interfaces that can make use of these constants

Interfaces vs Abstract Classes

- Purpose of interface similar to purpose of abstract class
- But … an interface is not a base class
  - It is not a class of any kind
- Use an abstract base class when
  - You need a method or private data field that classes will have in common
- Otherwise use an interface

Choosing Classes

- Look at a prospective system from a functional point of view
- Ask
  - What or who will use the system
  - What can each actor do with the system
  - Which scenarios involve common goals
- Use a case diagram

Choosing Classes

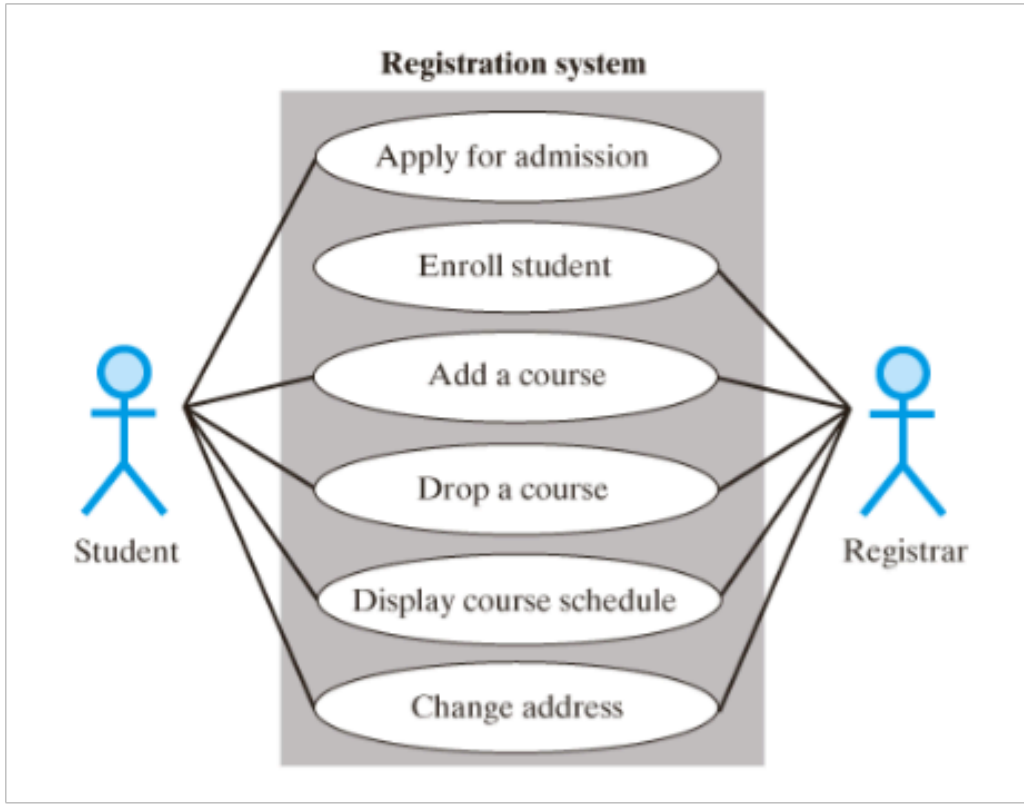


Fig. 1-16 A use case diagram for a registration system

Identifying Classes

- Describe the system
  - Identify nouns and verbs
- Nouns suggest classes
  - Students
  - Transactions
  - Customers
- Verbs suggest appropriate methods
  - Print an object
  - Post a transaction
  - Bill the customer

Identifying Classes

System: Registration
Use case: Add a course
Actor: Student
Steps:

1. Student enters identifying data.
2. System confirms eligibility to register.
   a. If ineligible to register, ask student to enter identification data again.
3. Student chooses a particular section of a course from a list of course offerings.
4. System confirms availability of the course.
   a. If course is closed, allow student to return to Step 3 or quit.
5. System adds course to student's schedule.
6. System displays student's revised schedule of courses.

Fig. 1-17 A description of a use case for adding a course

Reusing Classes

- Much software combines:
  - Existing components
  - New components
- When designing new classes
  - Plan for reusability in the future
  - Make objects as general as possible
  - Avoid dependencies that restrict later use by another programmer

Summary

- An Object is a program construction that contains data and methods
- A Class is a type or kind of Object
- Data and methods can be public or private

Summary

- A constructor allocates memory for the object and initialises the data fields
- A static field/method is associated with the Class and not the Object
- A package is a group of related classes

Summary

- Composition defines a ‘has a’ relationship between classes
- Inheritance groups classes that have common properties (an ‘is a’ relationship)
- Derived class methods can override base class methods
- Methods can be overloaded when two+ methods have the same name, but different parameters

Summary

- Protected methods can be accessed within its own class, derived class or package. Other classes CANNOT invoke it
- Every class is a descendant of the class Object
- An Abstract class has no instance and only acts as a base class

Summary

- Polymorphism is where an object decides at runtime which action of an overridden method to use
- Encapsulation is a design principal that hides details of class implementation (“Black Box”)
- Abstraction focuses on *what* not *how*
- An Interface declares methods that a class must implement and also data constants

Summary

- A class that implements an Interface must have an implements statement in the class definition
- A Java class can implement any number of Interfaces

Bibliography

- Frank M. Carrano & Walter Savitch, *"Data Structures and Abstractions with Java"*, Prentice Hall/Pearson Education, 2003
- David J. Barnes & Michael Kölling, "*Objects First with Java: A Practical Introduction using BlueJ"*, Prentice Hall / Pearson Education, 2006
- Eclipse Software Development Kit www.eclipse.org