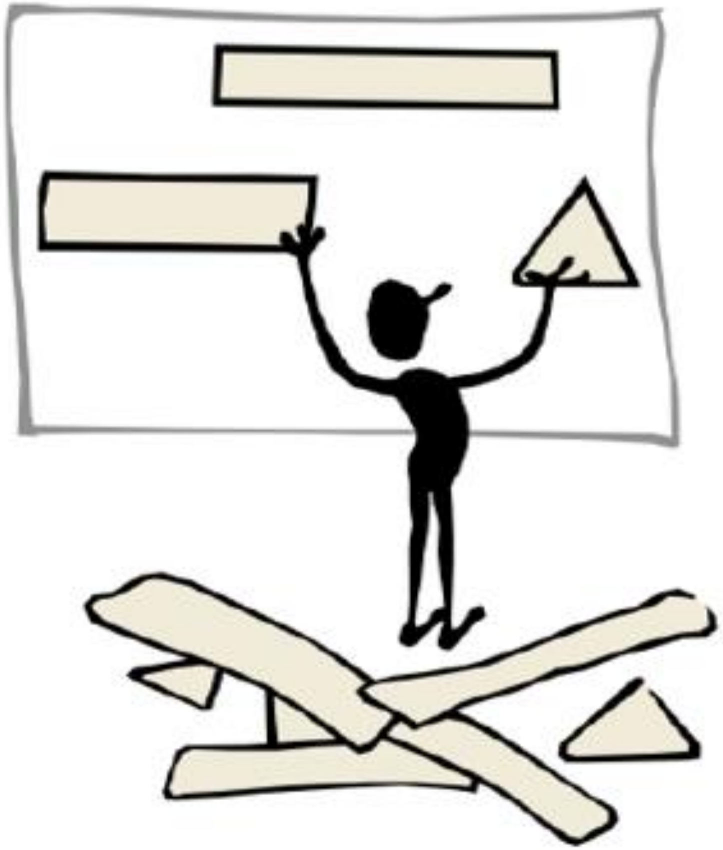
Chapter 2

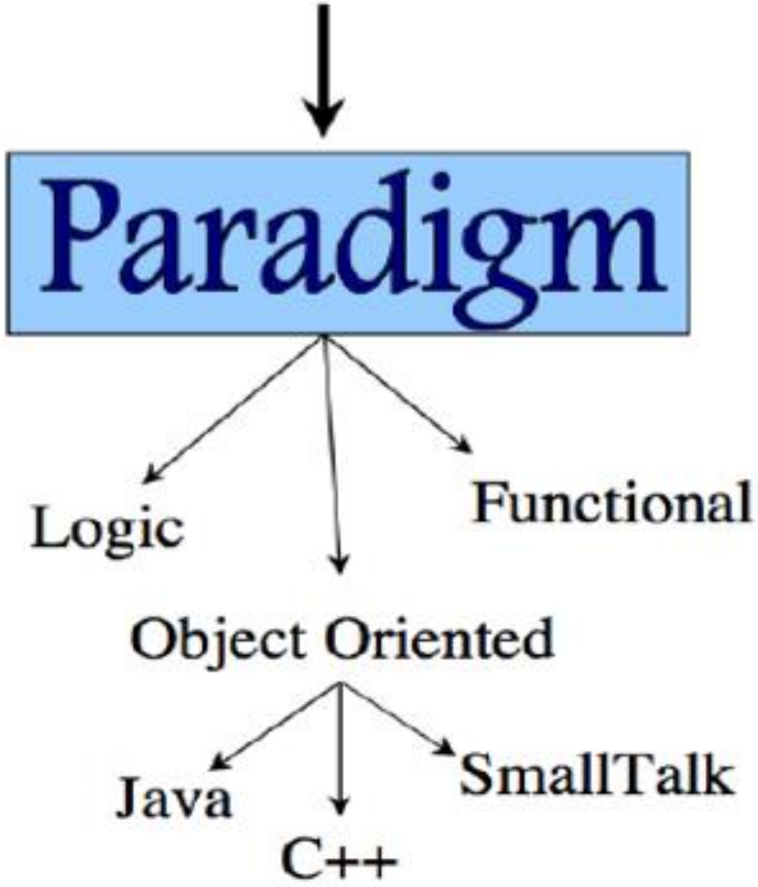
Object Oriented Design Concepts

Outline

- Abstract types
- Interfaces
- Polymorphism
- Delegation vs. sub classing (Inheritance)
- Generics

Software Design Paradigm





Paradigm

Logic

Functional

Object Oriented

Java

C++

SmallTalk

Functional Paradigm

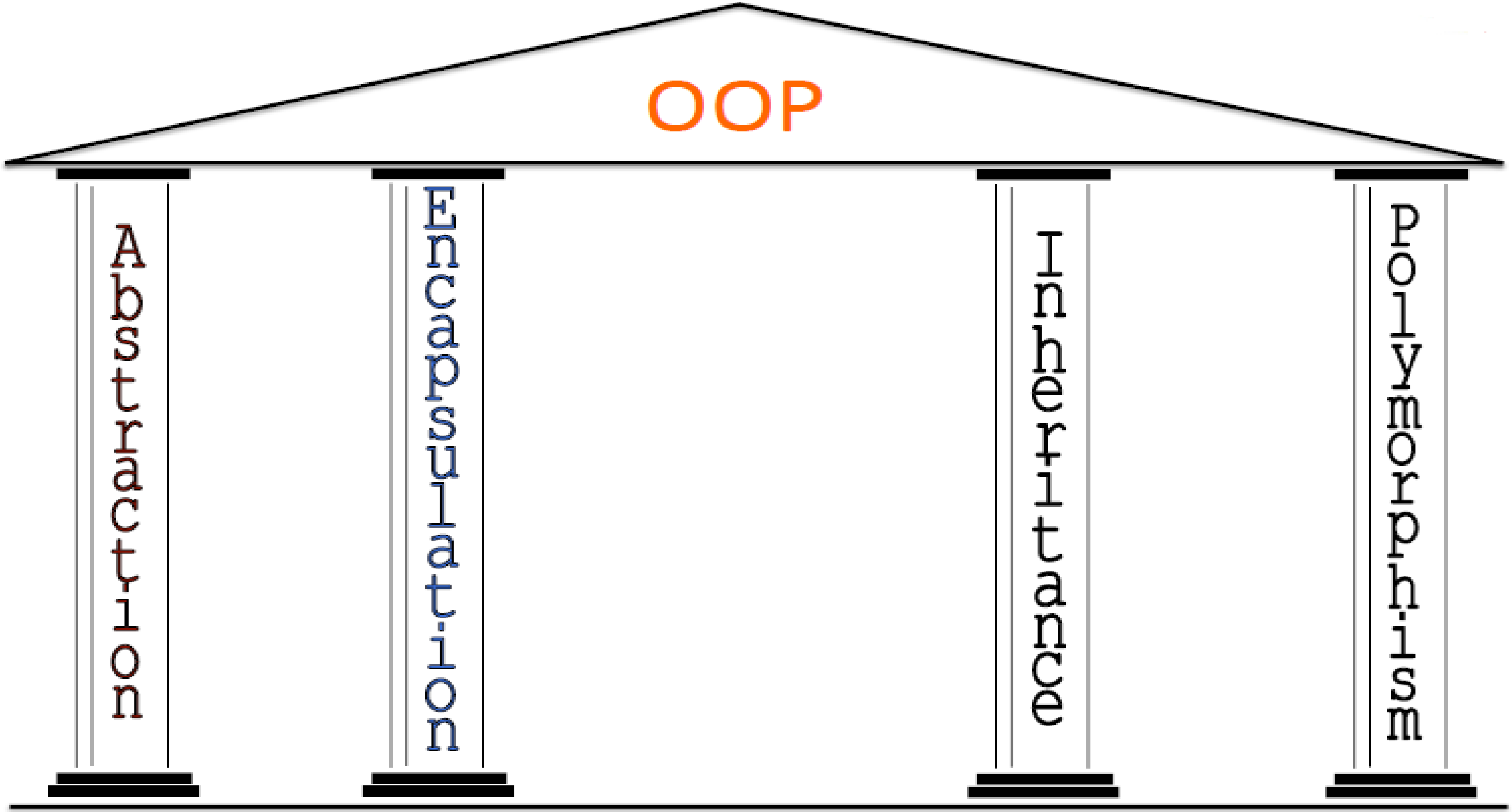
- We think in terms of functions acting on data
  - ✓**Abstraction**: think of the problem in terms of a process that solves it
  - ✓**Decomposition**: break your processing down into small manageable processing units(functions)
  - ✓**Organization**: set up functions so that they call each other (function calls, arguments etc.)
- **FIRST**: define your set of data structures(type etc.)
- **THEN**: define your set of functions acting upon the data structures

Object Oriented Paradigm

- We think in terms of objects interacting
  - **Abstraction**: think in terms of independent agents(objects) working together.
  - **Decomposition** : define the kinds of objects on which to split the global task
  - **Organization**: create the appropriate number of objects of each kind
  - **FIRST**: define the behavior and properties of objects of the different kinds we have defined
  - **THEN**: set up objects of each kind and put them to work

Object Oriented Scope

- **OBJECT-ORIENTED ANALYSIS:** Examines the **requirements of a system or a problem** from the perspective of **the classes and objects found in the vocabulary of the problem domain**.
- **OBJECT-ORIENTED DESIGN: Architectures** a system as made of **objects and classes, specifying their relationships** (like inheritance) and interactions.
- **OBJECT-ORIENTED PROGRAMMING:** A method of implementation in which programs are organized **as cooperative collections of objects**, each of which represents **an instance of some class**, and whose classes are all **members of a hierarchy of classes**.



OOP

Abstraction

Encapsulation

Inheritance

Polymorphism

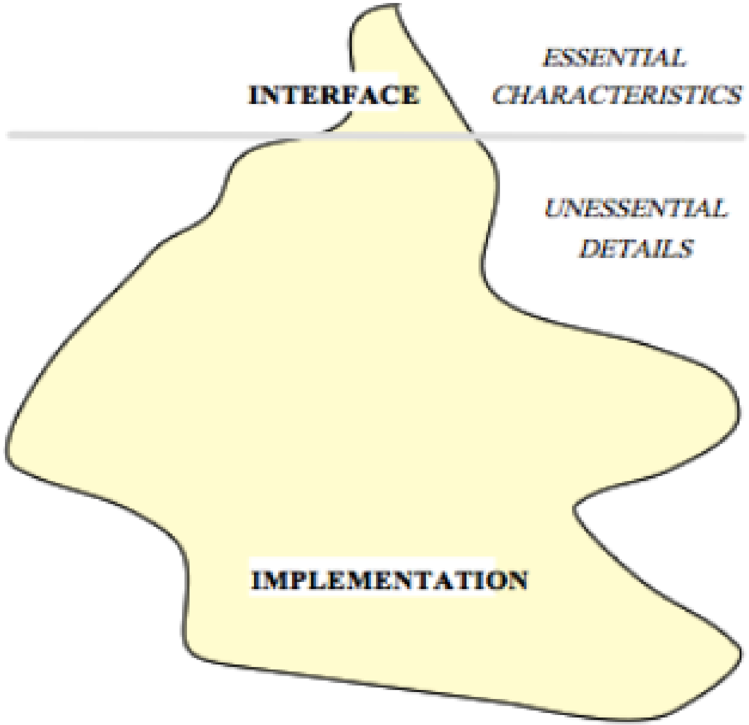
Abstract Data Types

- Data abstraction, or abstract data types, is a programming methodology where one defines not only the data structure to be used, but the processes to manipulate the structure like process abstraction, ADTs can be supported directly by programming languages
- To support it, there needs to be mechanisms for
  1) defining data structures
  2) encapsulation of data structures and their routines to manipulate the structures into one unit

     -by placing all definitions in one unit, it can be compiled at one time
  3) information hiding to protect the data structure from outside interference or manipulation
     - the data structure should only be accessible from code encapsulated with it so that the structure is hidden and protected from the outside
     - objects are one way to implement ADTs, but because objects have additional properties.

Abstraction

- Abstraction allows designers to focus on solving a problem without being concerned about irrelevant lower level details.
- Abstraction manages complexity by emphasizing on essential characteristics and leaving implementation details



Common types of abstraction

- **Procedural abstraction**

  e.g., closed subroutine

- **Data abstraction**

  e.g., ADT (Abstract Data Type) classes

- **Control abstraction**

  e.g., loops.

Abstraction (contd.)

- Allows postponement of certain design decisions that occur at various levels of analysis
- e.g.,
  - Representational and algorithmic considerations
  - Architectural and structural considerations
  - External environment and platform considerations
- **The two basic abstraction types:**
  - **Procedural abstraction**
    - Ø abstractions of the events in the system.
    - Ø consists of named sequence of events
  - **Data abstraction**
    - Ø named collection of data objects

Abstraction in Object Oriented Programming (OOP)

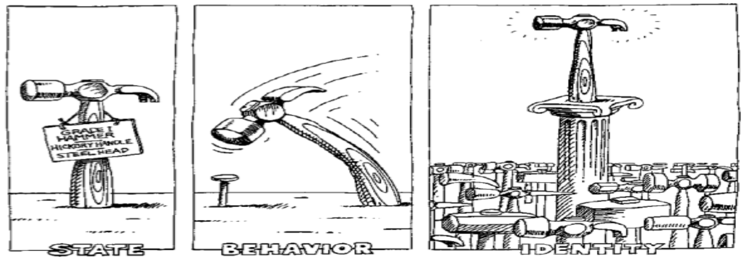
- In OOP, Objects are the main means of abstraction
- An object identifies specific entities.

  - An object is either an abstract (conceptual) or concrete entity.
- **An object is defined by:**
  - Attributes (data)
  - Operations
  - Identity
    - does not depend on the current value of the attributes
    - This never changes
- A **class** represents a set of objects that share a common structure and a common behavior

Cont’d

- **Examples:**
- **Time**
  - **Data**: (Hour, Minute, Second, Date)
  - **Operations**: add time interval, calculate difference from another period of time, etc.
  - **Acts (event)- for example: Measurement of a patients’ fever**
    - **Data**: temperature, Measured by, Time
    - **Operations**: Print, update, archive
- **At a given time, each object has:**
  - **State**: the current value of the attributes
  - **Behavior:** the set of operations they recognize, and the way they are interpreted
  - **Identity**: a constant that is determined independently the object is said to occupy a unique expanse in memory

➢ **Objects (state, behavior, identity)**



Abstract Classes & Interfaces

- Definitions
  - **Abstract methods** = Methods that are declared, with no implementation
  - **Abstract class** = A class with abstract methods, not meant to be instantiated
  - **Interface** = A named collection of method definitions (without implementations)
- **Examples**
  - Food is an abstract class. Can you make an instance of food? No, of course not. But you can make an instance of an apple or a steak or a peanut butter cup, which are types of food. Food is the abstract concept; it shouldn’t exist.
  - Skills are interfaces. Can you make an instance of a student, an athlete or a chef? No, but you can make an instance of a person, and have that person take on all these skills. Deep down, it’s still a person, but this person can also do other things, like study, sprint and cook.

Abstract Classes & Interfaces

- **Q: So what is the difference between an interface and an abstract class?**
- **A:**
  - An **interface** any methods, whereas an abstract class can
  - A class can **implement** **many interfaces** but can have only one superclass (abstract or not)
  - An interface is **not part of the class hierarchy**. Unrelated classes can implement the same interface
- **Syntax**:
  - **abstract class:**
  - **Public abstract class Food { }**

    public class Apple extends Food { … }
  - **interface:**

    public class Person implements Student, Athlete, Chef {….}

Abstract Classes & Interfaces…

- **Q: Why are they useful?**
- **A:** By leaving certain methods undefined, these methods can be implemented by several different classes, each in its own way.
- **Example:** Chess Playing Program

– an abstract class called ChessPlayer can have an abstract method makeMove(), ***extended*** differently by different subclasses.

```
public abstract class ChessPlayer {
<variable declarations>
<method declarations>
public void makeMove(); }
```

– an interface called ChessInterface can have a method called makeMove(), implemented differently by different classes.

```
public interface ChessInterface {
public void makeMove(); }
```

The Object Concept

- An object is an encapsulation of data.
- An object has:
  1. An identity (a unique reference)

     e.g.: Social security number (SSN), employee number, passport number, Student number

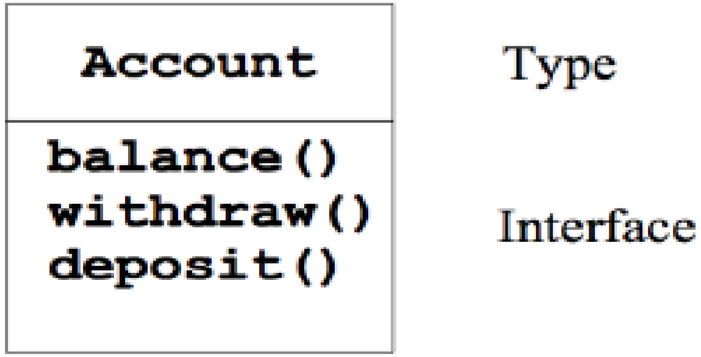
     Account a = new Account()
  2. State, also called characteristics (variables)

     e.g.: hungry, sad, drunk, running, alive
  3. Behavior (methods)

     e.g.: eat, drink, smile, kiss and wave.
- An object is an instance of a class.
- A class is oden called an Abstract Data Type (ADT).

Type and an Interface of Object

❑ An object has type and an interface.



Account

balance()
withdraw()
deposit()

Type

Interface

❑To get an object

```
Account  a= new Account();
Account  b= new Account();
```

❑To send a message

```
a.withdraw();
b.balance();
a.deposit();
```

The Object Concept

- A class is a collection of *objects* (or *values*) and a corresponding set of *methods.*
- A class encapsulates the data representation and makes data access possible at a higher level of abstraction.
- A class defines a template of the similar objects.

Class Vs. Object

Class

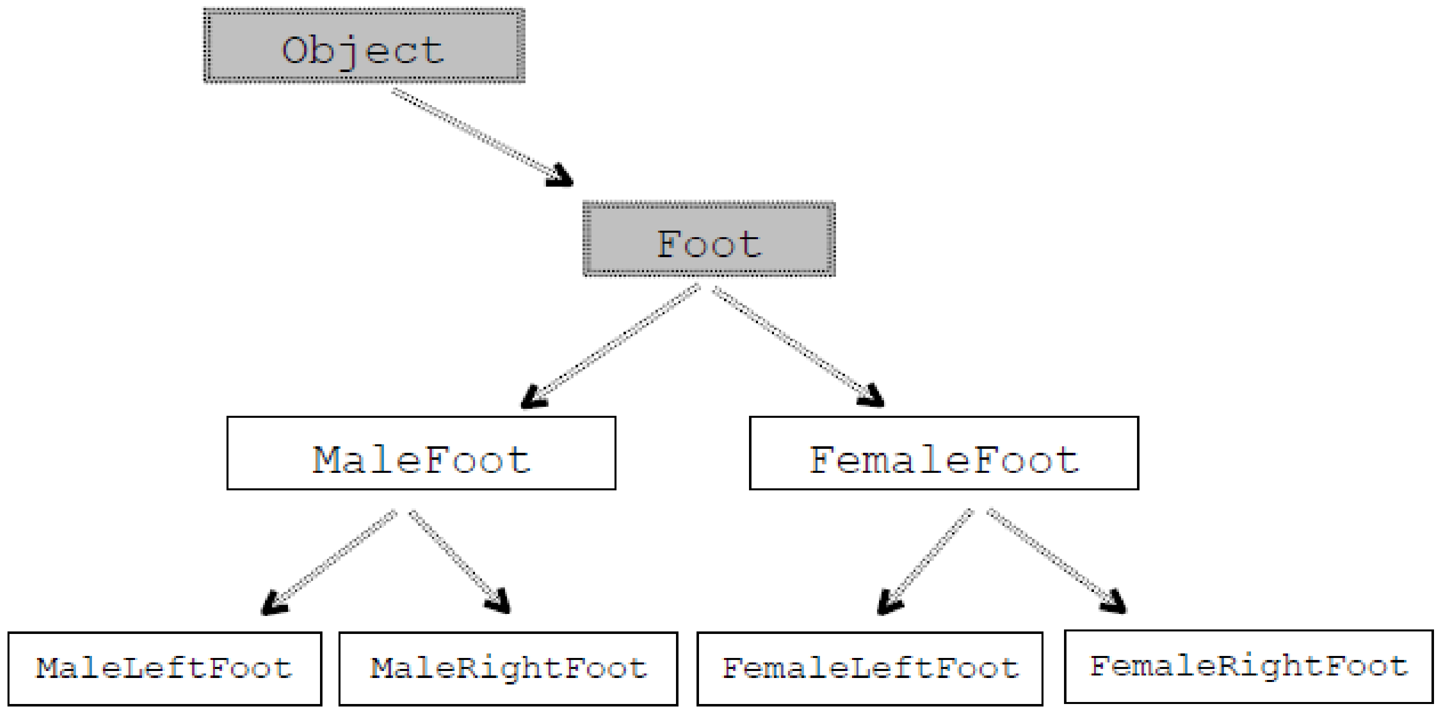
- A description of the *common properties* of a set of objects.
- A concept.
- A class is a part of a program.

Object

- A representation of the *properties* of a single instance.
- A real world entity.
- An object is part of data and a program execution.

Cont’d

- **Inheritance** may be used to define a hierarchy of classes in an application:



Every derivation should be an is-a relationship

Polymorphism

- *Polymorphism* comes from Greek meaning "**many forms**."
- In Java, polymorphism refers to the dynamic binding mechanism that determines which method definition will be used when a method name has been **overridden**.
- Thus, polymorphism refers to dynamic binding.
- The ability of a function to respond differently when supplied with arguments that are objects of different types is called ***functional overloading***.
- Polymorphism ensures that the appropriate method is called for an object of a specific type when the object is disguised as a more general type.
- A ***polymorphic reference*** is a variable that can refer to **different types of objects** at different points in time.

Polymorphism (Cont'd)

- Can treat an object of a subclass as an object of its superclass

– A reference variable of a superclass type can point to an object of its subclass

```
Person name, nameRef;
PartTimeEmployee employee, employeeRef;
name = new Person("John", "Blair");
employee = new PartTimeEmployee("Susan", "Johnson",
12.50, 45);
nameRef = employee;
System.out.println("nameRef: " + nameRef);
nameRef: Susan Johnson wages are: $57.50
```

Polymorphism (Cont'd)

- Late binding or dynamic binding (run-time binding):
  - Method to be executed is determined at execution time, not compile time
- **Polymorphism**: to assign multiple meanings to the same method name
- Implemented using late binding
- **Polymorphism**: Enables "programming in the general"

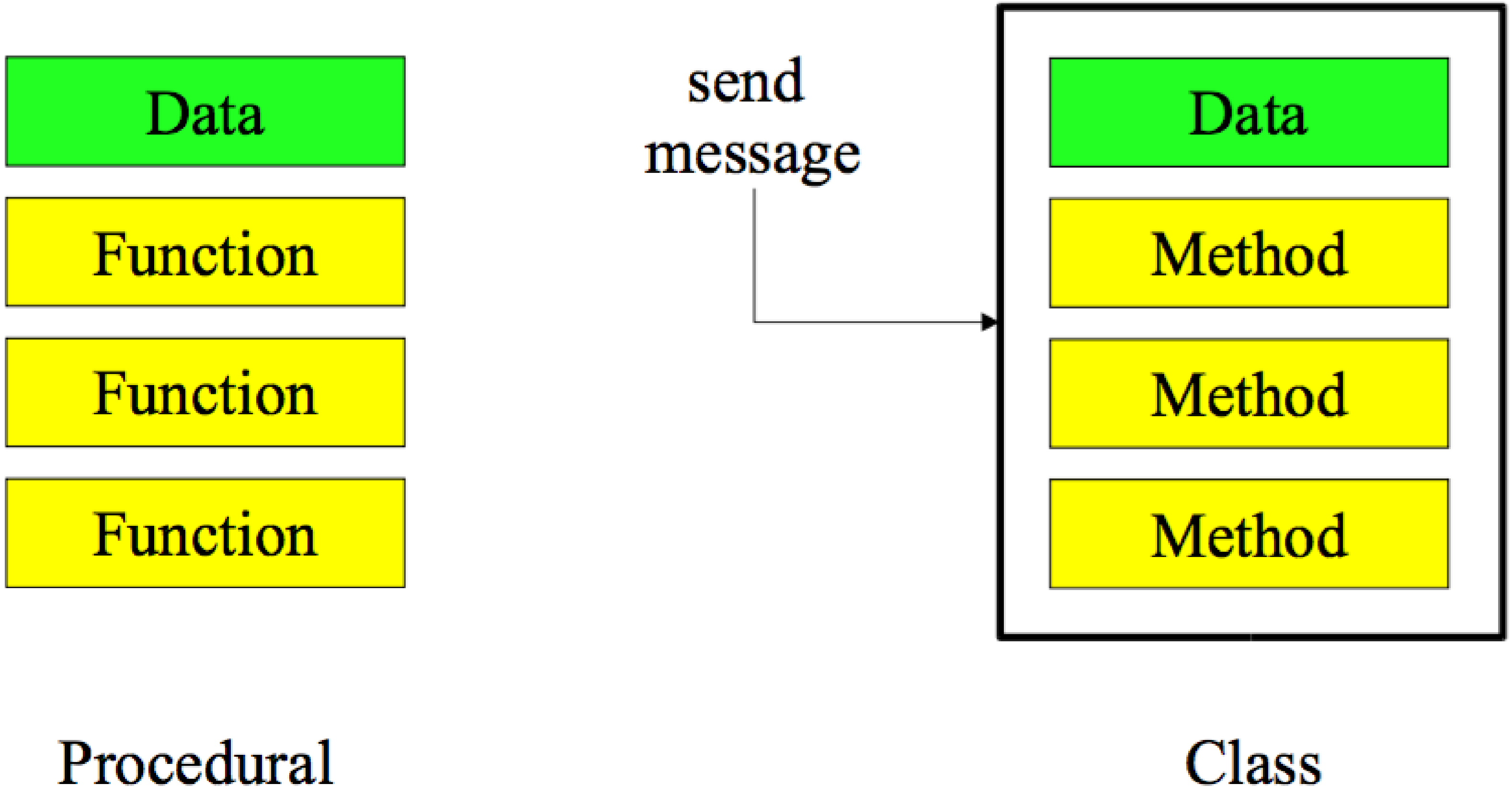
  --The same invocation can produce "many forms" of results
- Polymorphism promotes **extensibility** of the design in OOD.

Polymorphism (Cont’d)

- When a program invokes a method through a superclass variable,
  - the correct subclass version of the method is called,
  - based on the type of the reference stored in the superclass variable
- The same method name and signature can cause different actions to occur,
  - depending on the type of object on which the method is invoked

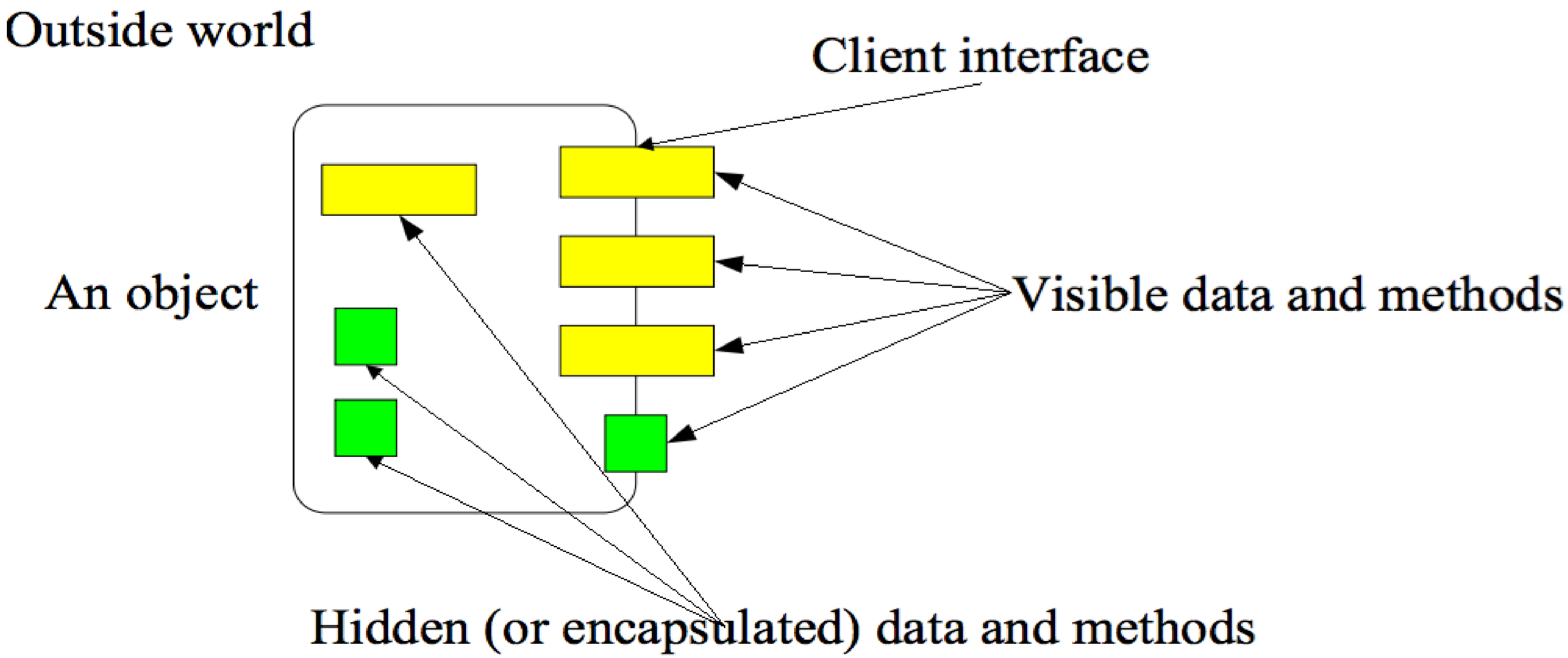
Encapsulation and Information Hiding

- Data can be encapsulated such that it is invisible to the “outside world”.
- Data can only be accessed via methods.



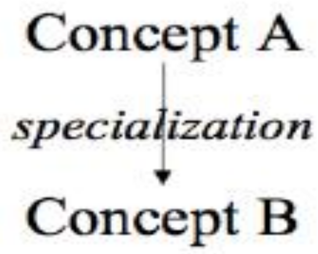
Encapsulation and Information Hiding…Cont..

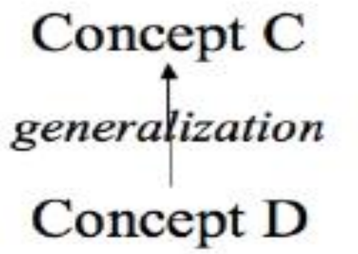
- What the “outside world” cannot see it cannot depend on!
- The object is a “fire-wall” between the object and the “outside world”.
- The hidden data and methods can be changed without affecting the “outside world”.

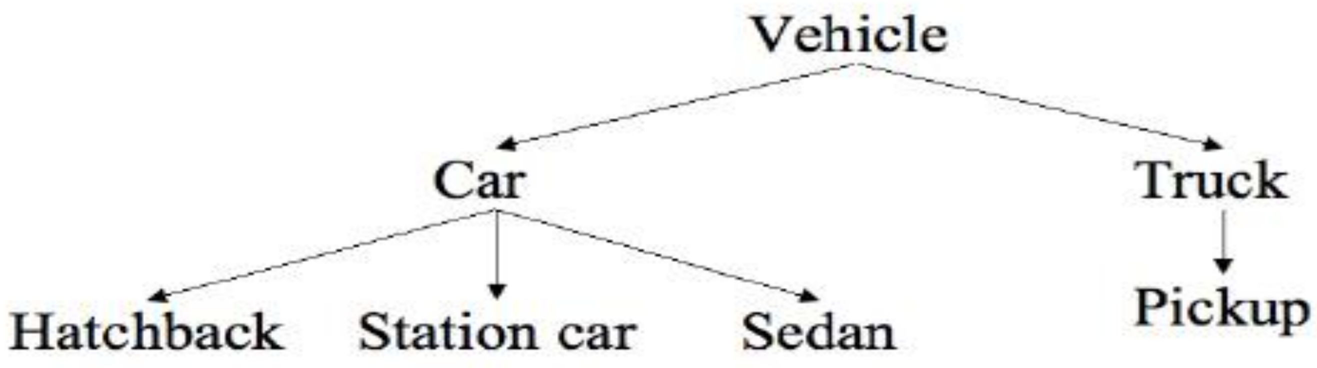


Generalization and Specialization

- Generalization creates a concept with a broader scope.
- Specialization creates a concept with a narrower scope.
- Reusing the interface!

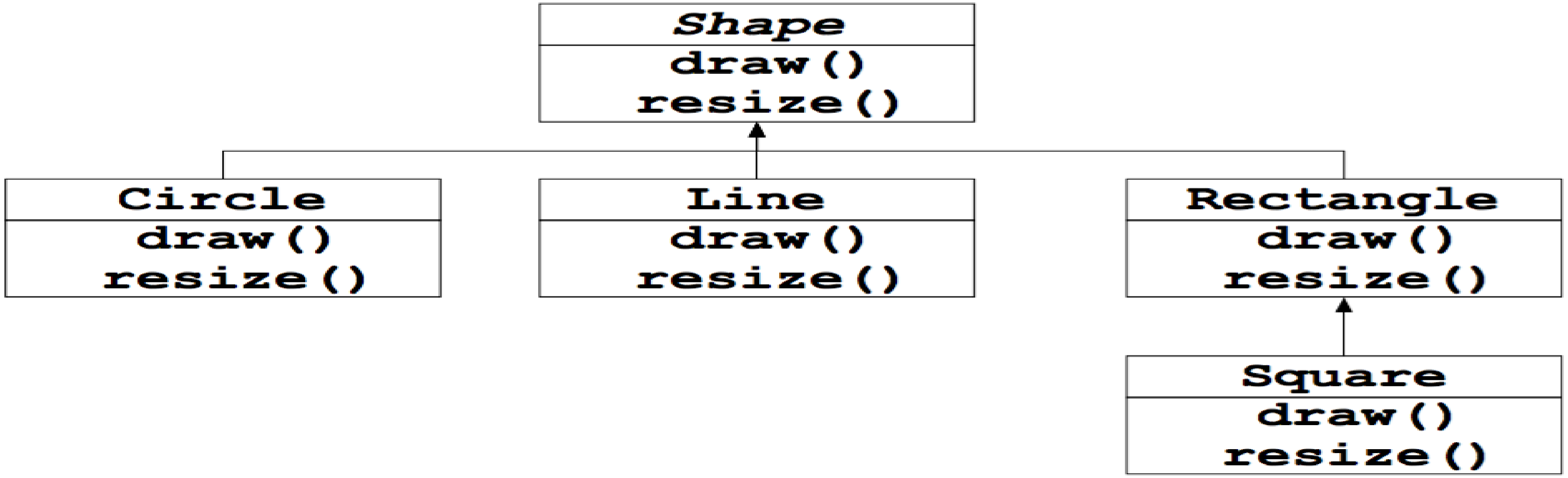






Generalization and Specialization, Example

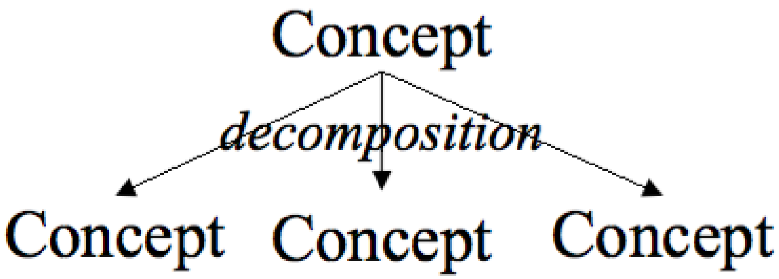
- *Inheritance*: get the interface from the general class.
- Objects related by inheritance are all of the same type.

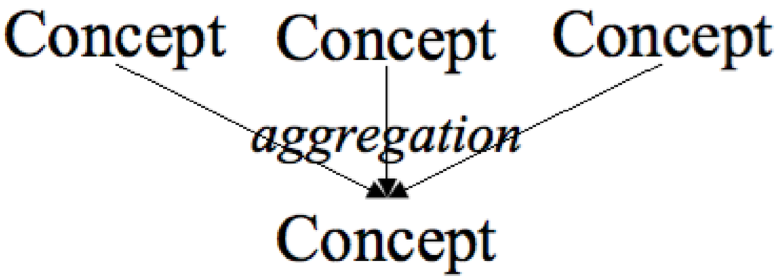


Square "is-a" Shape or **Square "is-like-a" Shape**

Aggregation and Decomposition

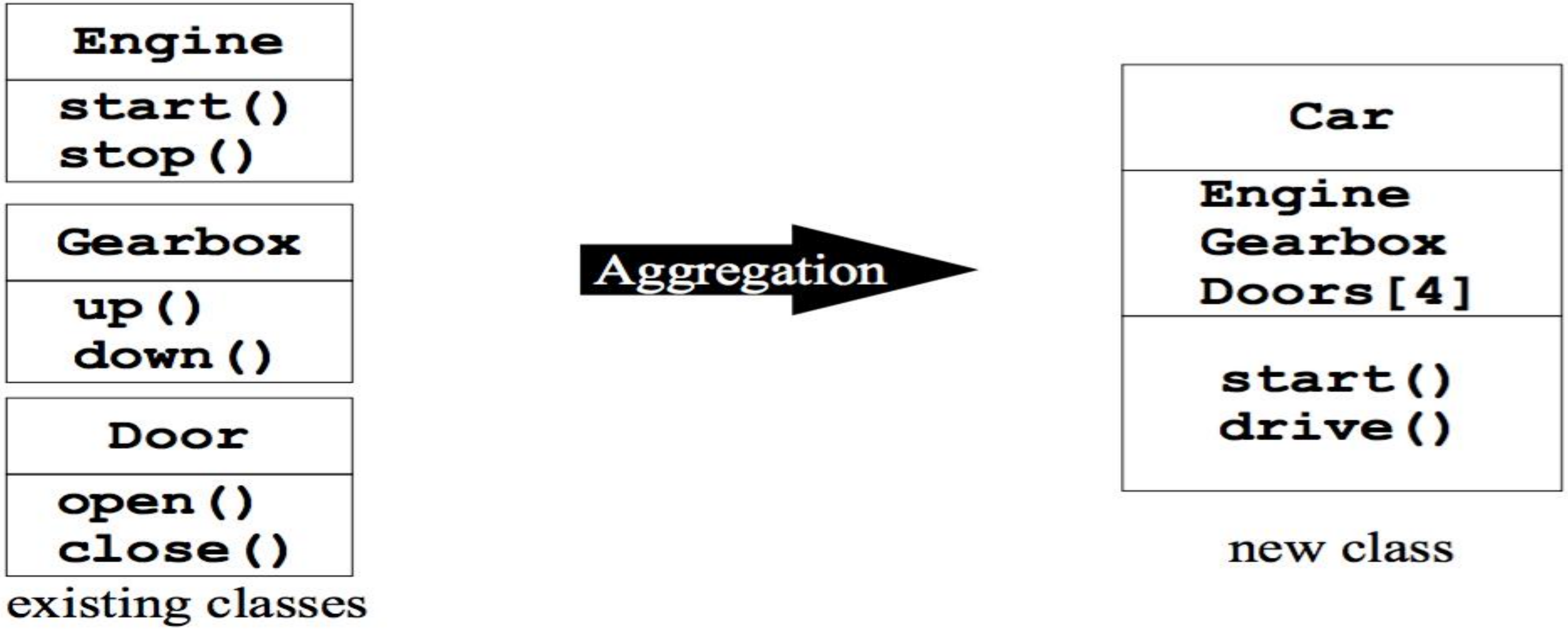
- An aggregation consists of a number of (sub-)concepts which collectively is considered a new concept.
- A decomposition splits a single concept into a number of (sub-)concepts.





Aggregation and Decomposition…

- **Idea**: make new objects by combining existing objects.
- *Reusing the implementation!*



Car “has-a” **Gearbox** and **Car** “has-an” **Engine**

What is Generics?

- Collections can store Objects of any Type
- Generics restricts the Objects to be put in a collection
- Generics ease identification of runtime errors at compile time

How is Generics useful?

- Consider this code snippet

```
List v = new ArrayList();
v.add(new String("test"));
Integer i = (Integer)v.get(0); // Runtime error .
Cannot cast from String to Integer
```

NOTE: This error comes up only when we are executing the program and not during compile time.

Thank You !