



Object Oriented Concepts and Terminology

Objectives

After completing this lesson, you should be able to do the following:

- Describe the important object-oriented (OO) concepts
- Describe the fundamental OO terminology

Examining Object Orientation

OO concepts affect the whole development process:

- Humans think in terms of nouns (objects) and verbs (behaviors of objects).
- With OOSD, both problem and solution domains are modeled using OO concepts.
- The *Unified Modeling Language (UML)* is a *de facto* standard for modeling OO software.
- OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.

Examining Object Orientation

“Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the objects types than on the actions.”

OO concepts affect the following issues:

- Software complexity
- Software decomposition
- Software costs

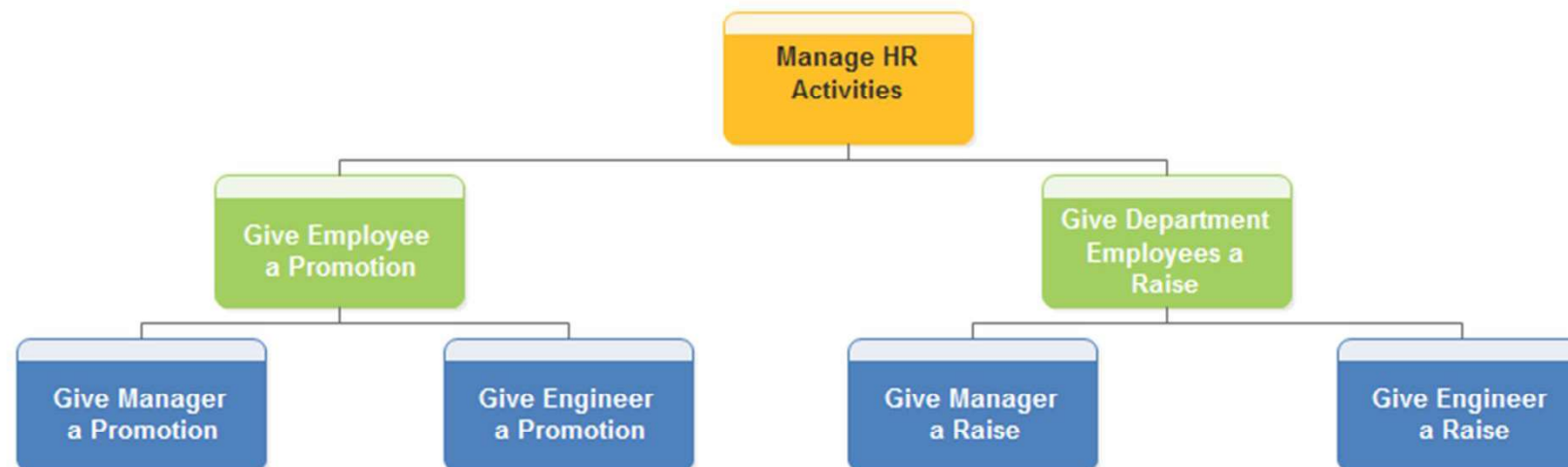
Software Complexity

Complex systems have the following characteristics:

- They have a *hierarchical structure*.
- The choice of *which components are primitive in the system* is arbitrary.
- A system can be split by intra- and inter-component relationships. This *separation of concerns enables you to study each part in relative isolation*.
- Complex systems are usually composed of only a *few types of components in various combinations*.
- A successful, complex system invariably *evolves from a simple working system*.

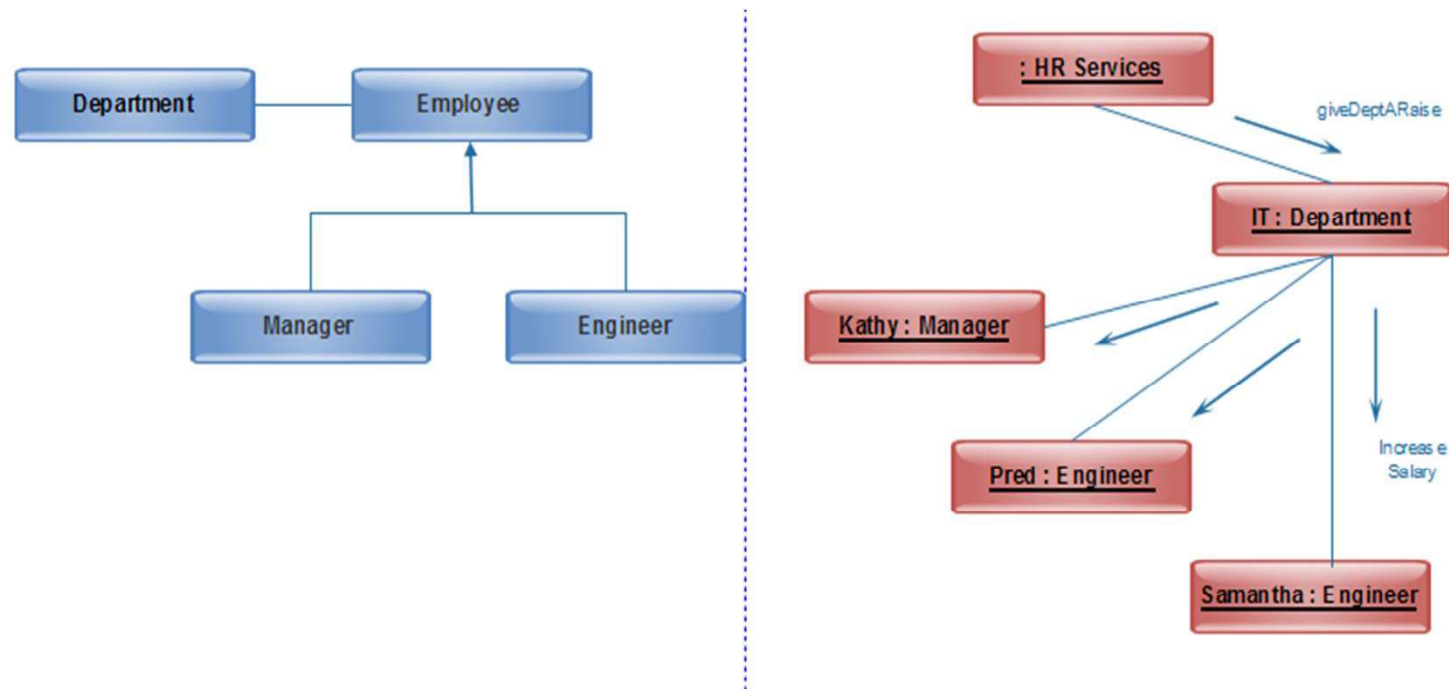
Software Decomposition

- In the Procedural paradigm, software is decomposed into a hierarchy of procedures or tasks.



Software Decomposition

- In the OO paradigm, software is decomposed into a hierarchy of interacting components (usually objects).



Software Costs

Development:

- OO principles provide a natural technique for modeling business entities and processes from the early stages of a project.
- OO-modeled business entities and processes are easier to implement in an OO language.

Maintenance:

- Changeability, flexibility, and adaptability of software is important to keep software running for a long time.
- OO-modeled business entities and processes can be adapted to new functional requirements.

Comparing the Procedural and OO

	Procedural Paradigm	OO Paradigm
Organizational structure	<p>Focuses on hierarchy of procedures and subprocedures</p> <p>Data is separate from procedures</p>	<p>Network of collaborating objects</p> <p>Methods (processes) are often bound together with the state (data) of the object</p>
Protection against modification or access	Data is difficult to protect against inappropriate modifications or access when it is passed to or referenced by many different procedures.	The data and internal methods of objects can be protected against inappropriate modifications or access by using encapsulation.

	Procedural Paradigm	OO Paradigm
Ability to modify software	Can be expensive and difficult to make software that is easy to change, resulting in many "Brittle" systems	Robust software that is easy to change, if written using good OO principles and patterns
Reuse	Reuse of methods is often achieved by copy-and-paste or 1001 parameters.	Reuse of code by using generic components (one or more objects) with well-defined interfaces. This is achieved by extension of classes (or interfaces) or by composition of objects.

	Procedural Paradigm	OO Paradigm
Configuration of special cases	Often requires if or switch statements. Modification is risky because it often requires altering existing code. So, modifications must be done with extreme care apart from requiring extensive regression testing. These factors make even minor changes costly to implement.	Polymorphic behavior can facilitate the possibility of modifications being primarily additive, subtractive, or substitution of whole components (one or more objects); thereby, reducing the associated risks and costs.

Surveying the Fundamental OO Concepts

- Objects and Classes
- Abstraction, Encapsulation , Inheritance and Polymorphism
- Interfaces
- Cohesion and Coupling
- Class associations and object links
- Delegation

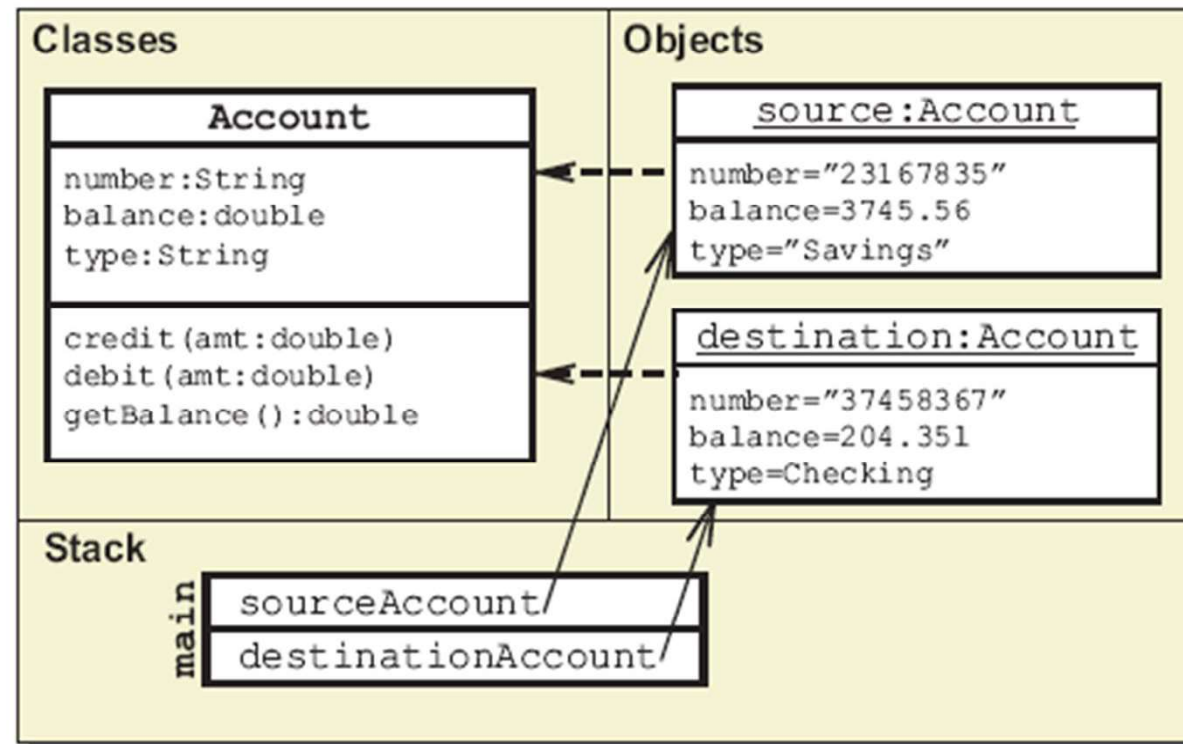
Classes

A class is a blueprint or prototype from which objects are created.

Classes provide:

- The metadata for attributes
- The signature for methods
- The implementation of the methods (usually)
- The constructors to initialize attributes at creation time

Classes: Example



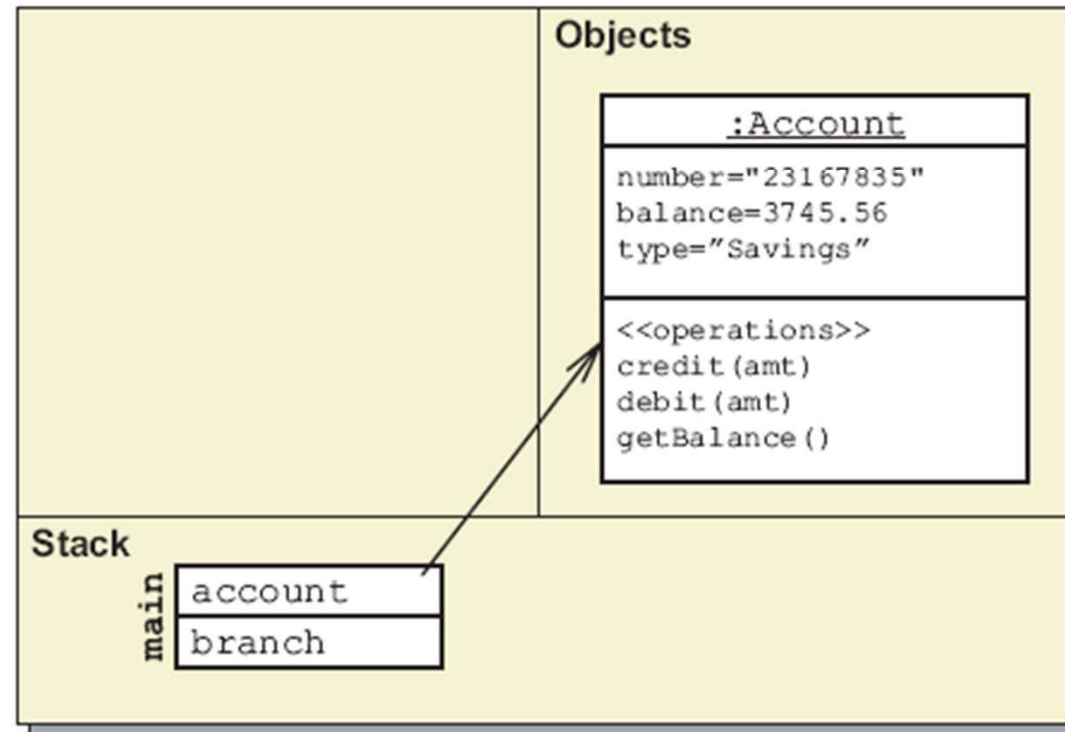
Objects

object = state + behavior

An object is a real-world element in an object-oriented environment that may have a physical or a conceptual existence. Each object has:

- Identity that distinguishes it from other objects in the system.
- State that determines the characteristic properties of an object as well as the values of the properties that the object holds.
- Behavior that represents externally visible activities performed by an object in terms of changes in its state.

Objects: Example



Abstraction

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

Abstraction: Example

Engineer
fname:String lname:String salary:Money
increaseSalary(amt) designSoftware() implementCode()

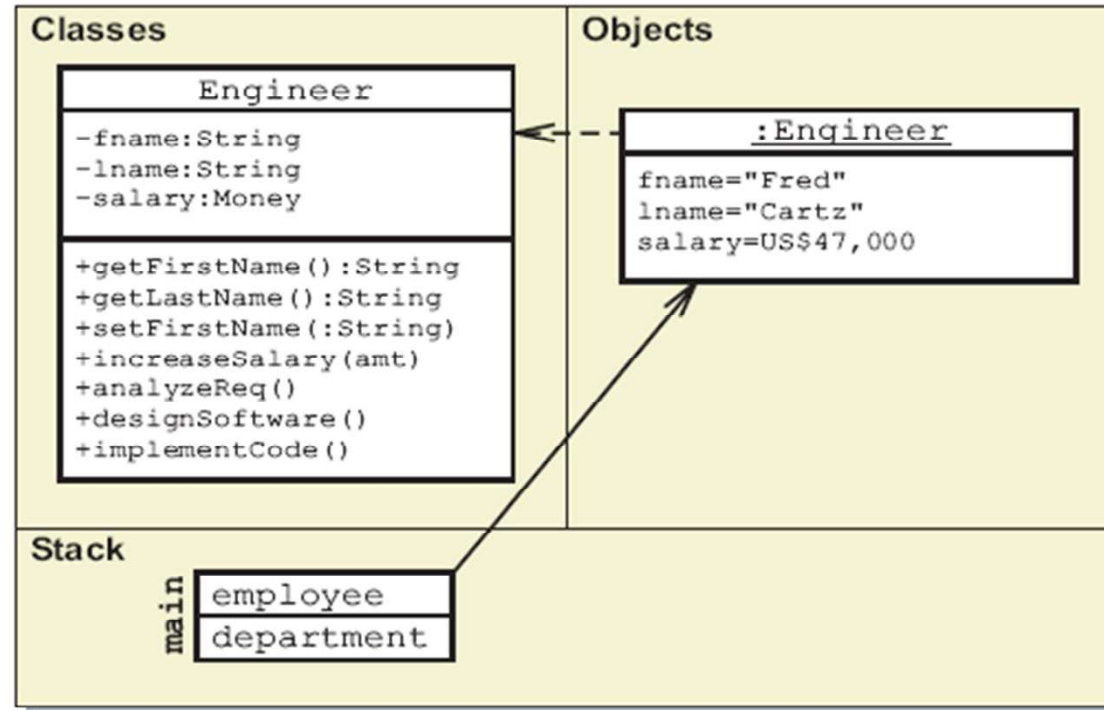
Engineer
fname:String lname:String salary:Money fingers:int toes:int hairColor:String politicalParty:String
increaseSalary(amt) designSoftware() implementCode() eatBreakfast() brushHair() vote()

Encapsulation

Encapsulation means “to enclose in or as if in a capsule”

- Encapsulation is essential to an object. An object is a capsule that holds the object's internal state within its boundary.
- In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “*hide* implementation details behind a set of non-private methods”.

Encapsulation: Example



✗ `name = employee.fname;`
✗ `employee.fname = "Samantha";`

✓ `name = employee.getFirstName();`
✓ `employee.setFirstName("Samantha");`

Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.”

Features of inheritance:

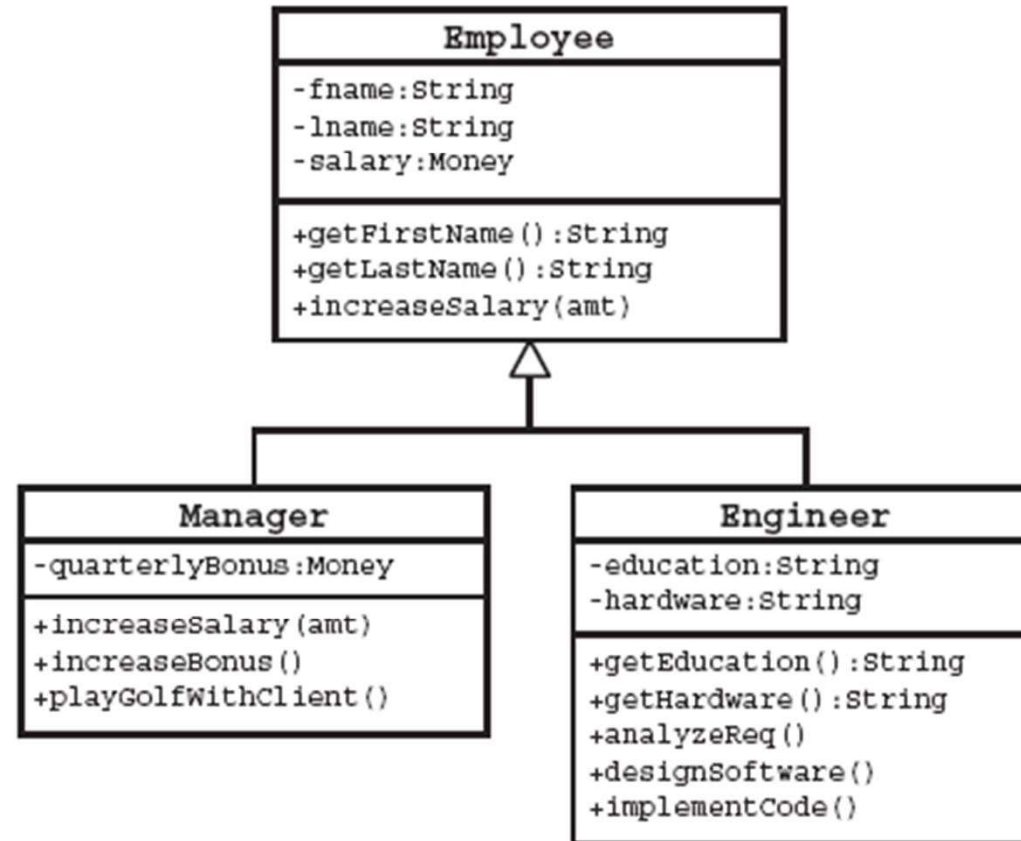
- Attributes and methods from the superclass are included in the subclass.
- Subclass methods can override superclass methods. • The following conditions must be true for the inheritance relationship to be plausible:
- A subclass object *is a (is a kind of) the superclass object*.
- Inheritance should conform to Liskov’s Substitution Principle (LSP).

Inheritance

Specific OO languages allow either of the following:

- Single inheritance, which allows a class to directly inherit from only one superclass (for example, Java).
- Multiple inheritance, which allows a class to directly inherit from one or more superclasses (for example, C++).

Inheritance: Example



Abstract Classes

A class that contains one or more abstract methods, and therefore can never be instantiated.

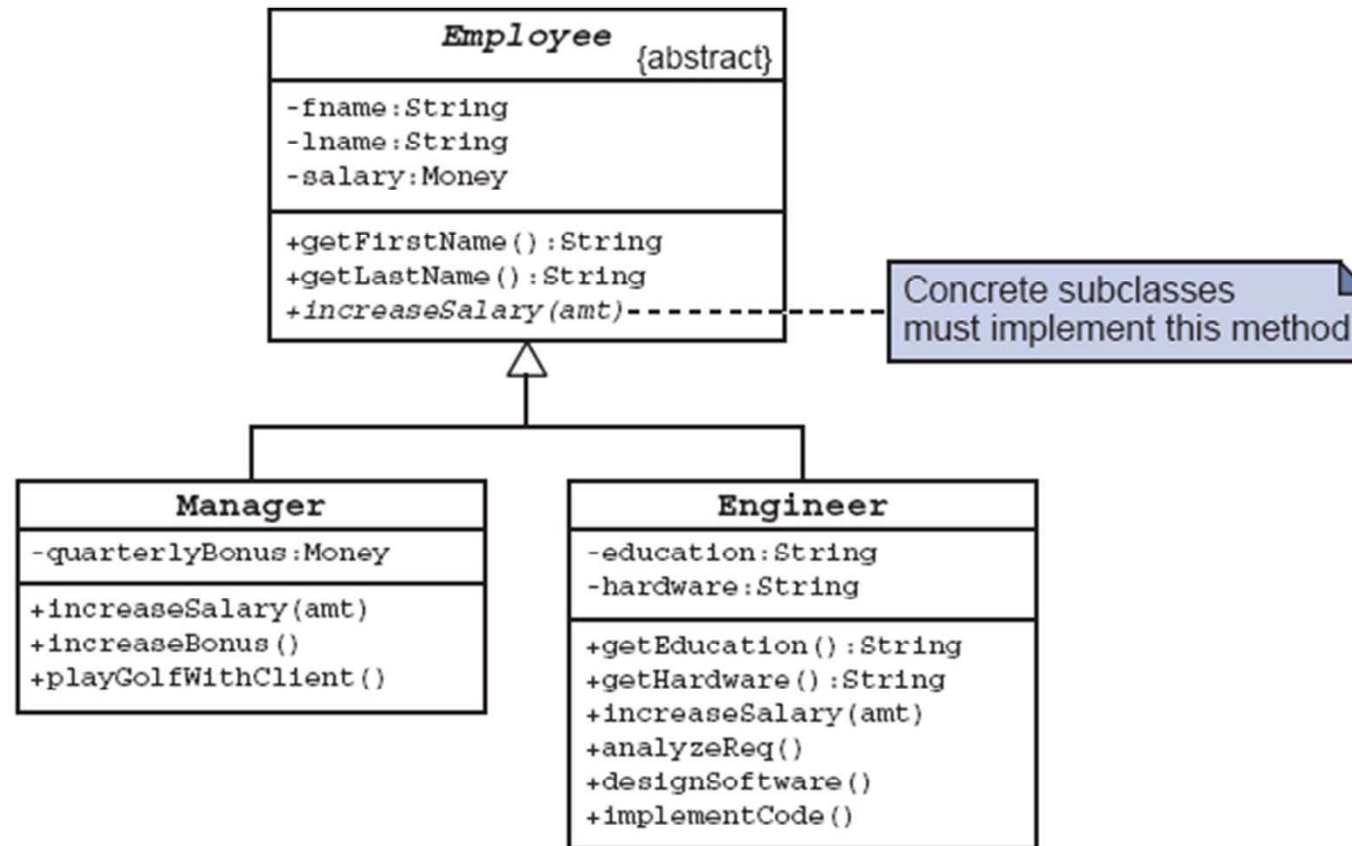
Features of an abstract class:

- Attributes are permitted.
- Methods are permitted and some might be declared abstract.
- Constructors are permitted, but no client may directly instantiate an abstract class.

Cont...

- Subclasses of abstract classes must provide implementations of all abstract methods; otherwise, the subclass must also be declared abstract.
- In the UML, a method or a class is denoted as abstract by using italics, or by appending the method name or class name with {abstract}.

Abstract Classes: Example



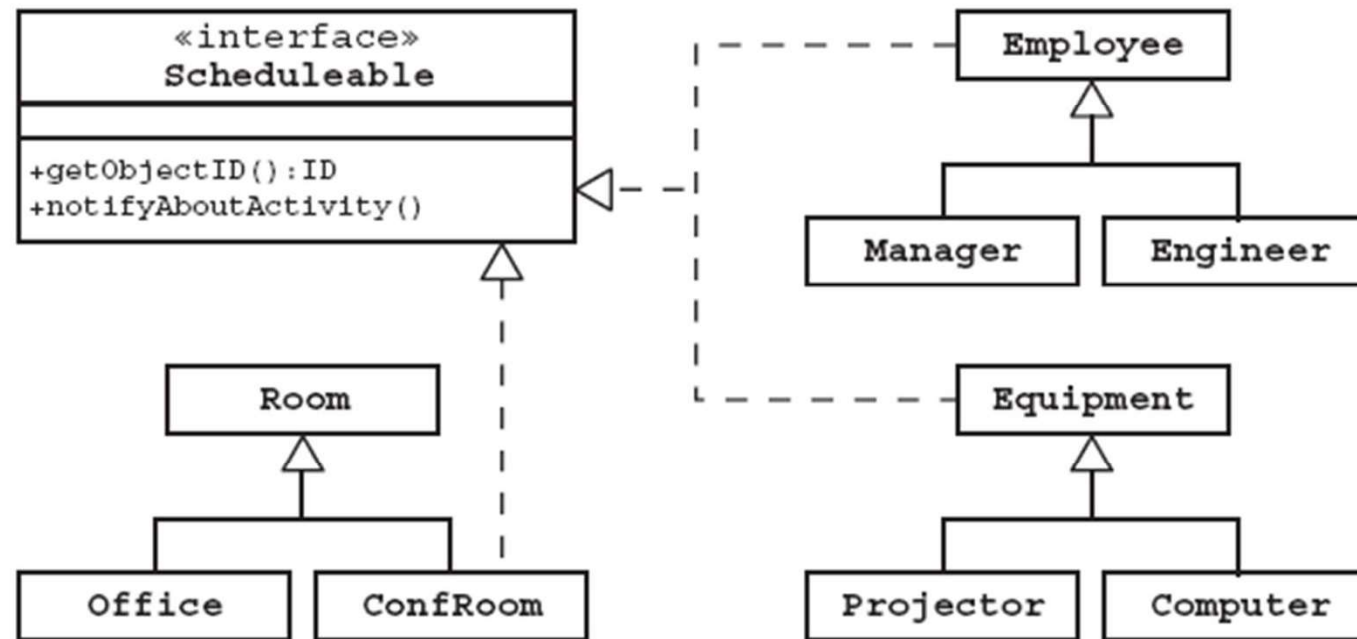
Interfaces

Features of Java technology interfaces:

- Attributes are not permitted (except constants).
- Methods are permitted, but they must be abstract.
- Constructors are not permitted.
- Subinterfaces may be defined, forming an inheritance hierarchy of interfaces.

A class may implement one or more interfaces.

Interfaces: Example



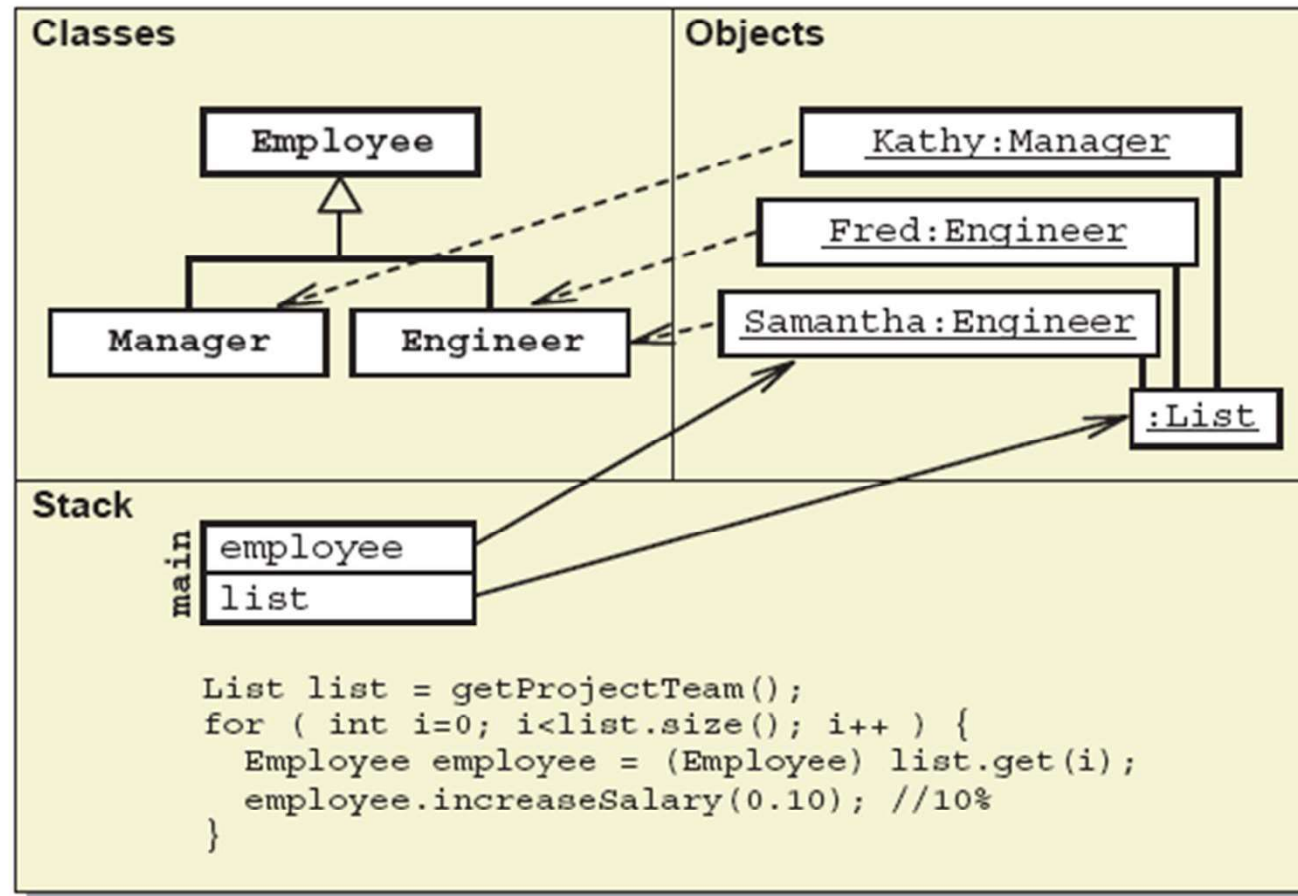
Polymorphism

Polymorphism is “a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass [type].”

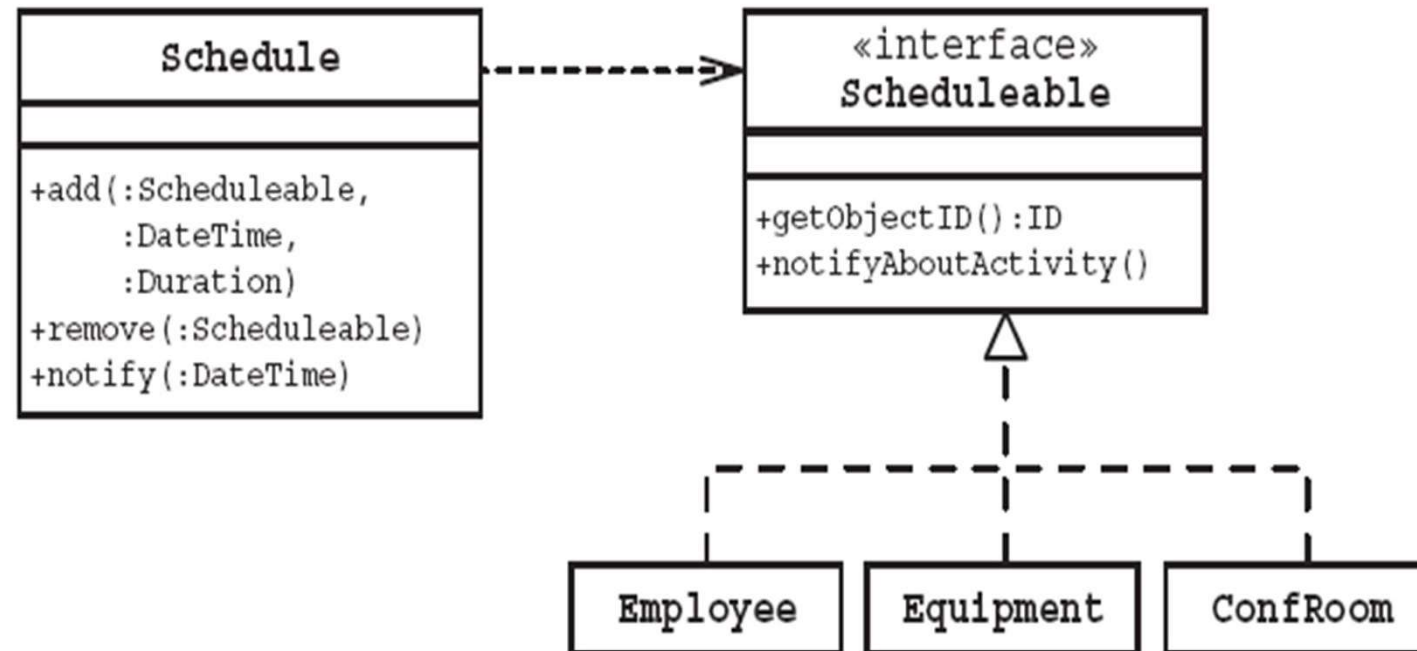
Aspects of polymorphism:

- A variable can be assigned different types of objects at runtime provided they are a subtype of the variable's type.
- Method implementation is determined by the type of object, not the type of the declaration (dynamic binding).
- Only method signatures defined by the variable type can be called without casting.

Polymorphism: Example

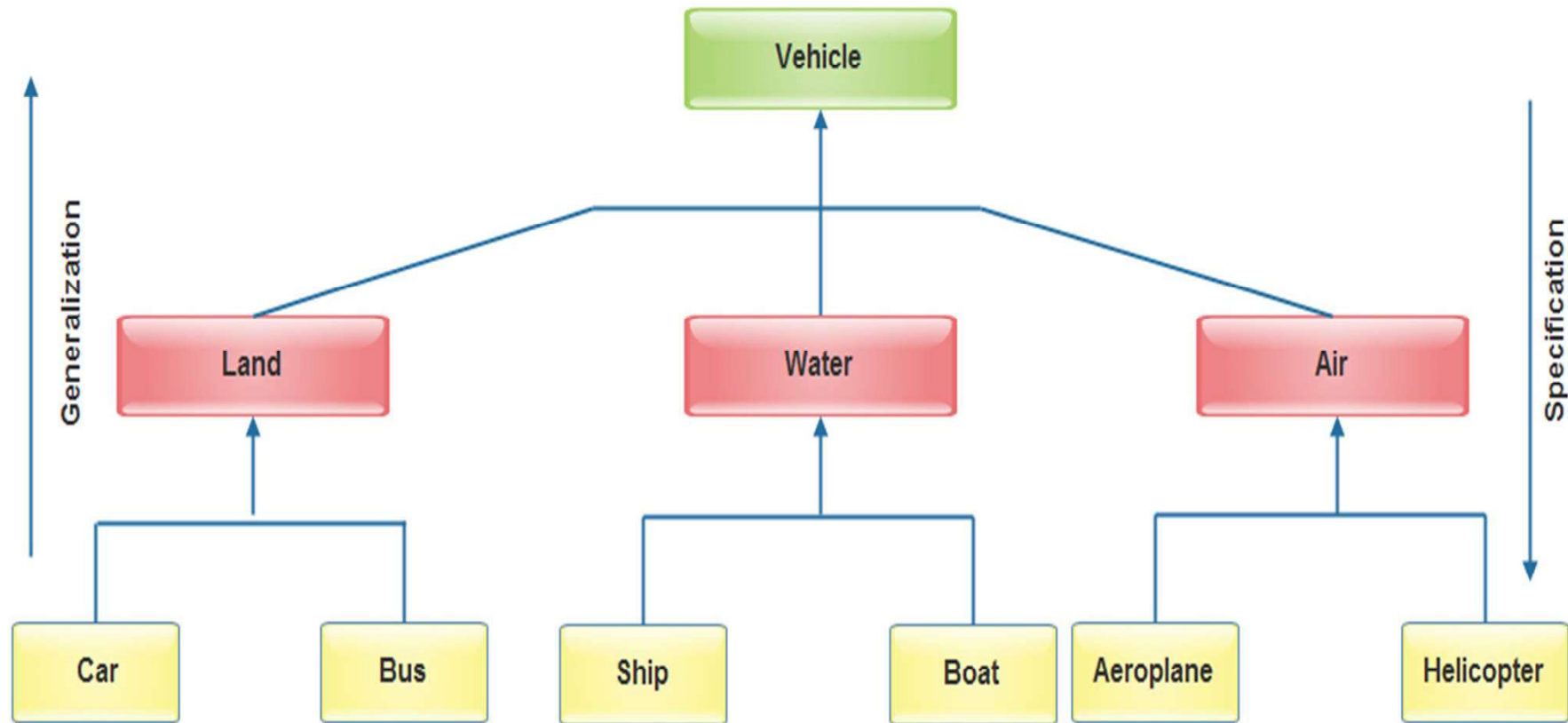


Example : In Context to Interfaces



Generalization and Specialization

- The common characteristics of classes are combined to form a class in a higher level of hierarchy, i.e., subclasses are combined to form a generalized super-class. It represents an “is – a – kind – of” relationship.
- Specialization is the reverse process of generalization. Here, the distinguishing features of groups of objects are used to form specialized classes from existing classes. It can be said that the subclasses are the specialized versions of the super-class.



Links and Association

- A link represents a connection through which an object collaborates with other objects.
- A link depicts the relationship between two or more objects.
- Association is a group of links having common structure and common behavior.
- Association depicts the relationship between objects of one or more classes. A link can be defined as an instance of an association.

Cohesion

In software, the concept of cohesion refers to how well a given component or method supports a single purpose.

- Low cohesion occurs when a component is responsible for many unrelated features.
- High cohesion occurs when a component is responsible for only one set of related features.
- A component includes one or more classes. Therefore, cohesion applies to a class, a subsystem, and a system.
- Cohesion also applies to other aspects including methods and packages.
- Components that do everything are often described with the Anti-Pattern term of Blob components.

Cohesion: Example

Low Cohesion

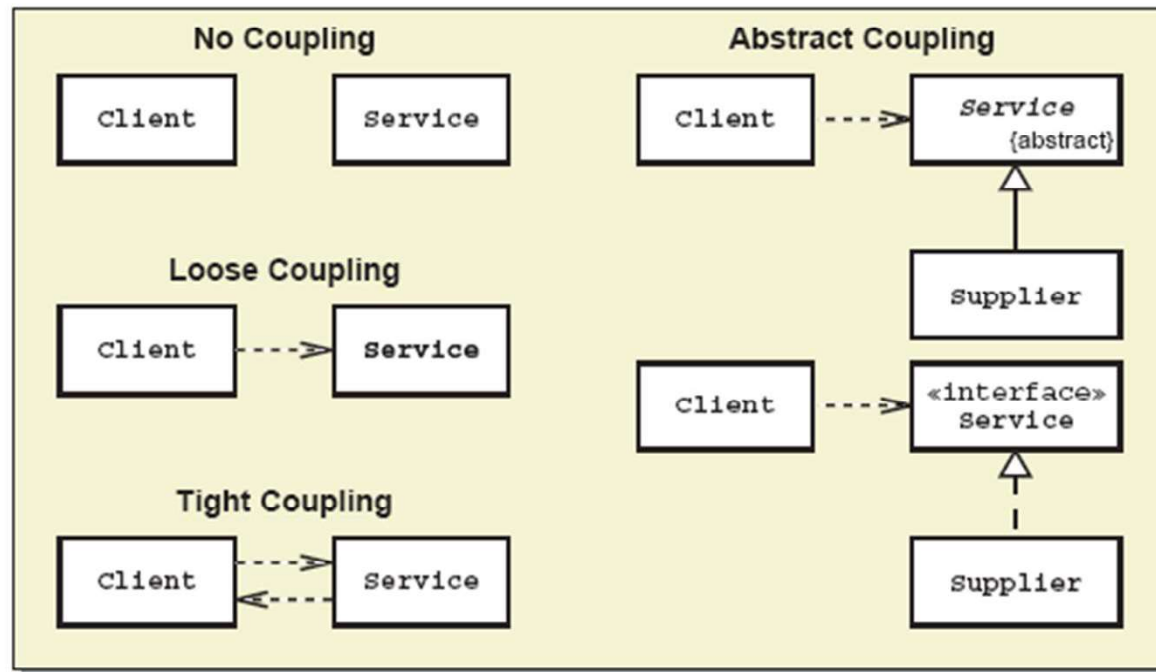
SystemServices
makeEmployee makeDepartment login logout deleteEmployee deleteDepartment retrieveEmpByName retrieveDeptByID

High Cohesion

LoginService
login logout
EmployeeService
makeEmployee deleteEmployee retrieveEmpByName
DepartmentService
makeDepartment deleteDepartment retrieveDeptByID

Coupling

- Coupling is “the degree to which classes within our system are dependent on each other.”



Class Associations and Object Links

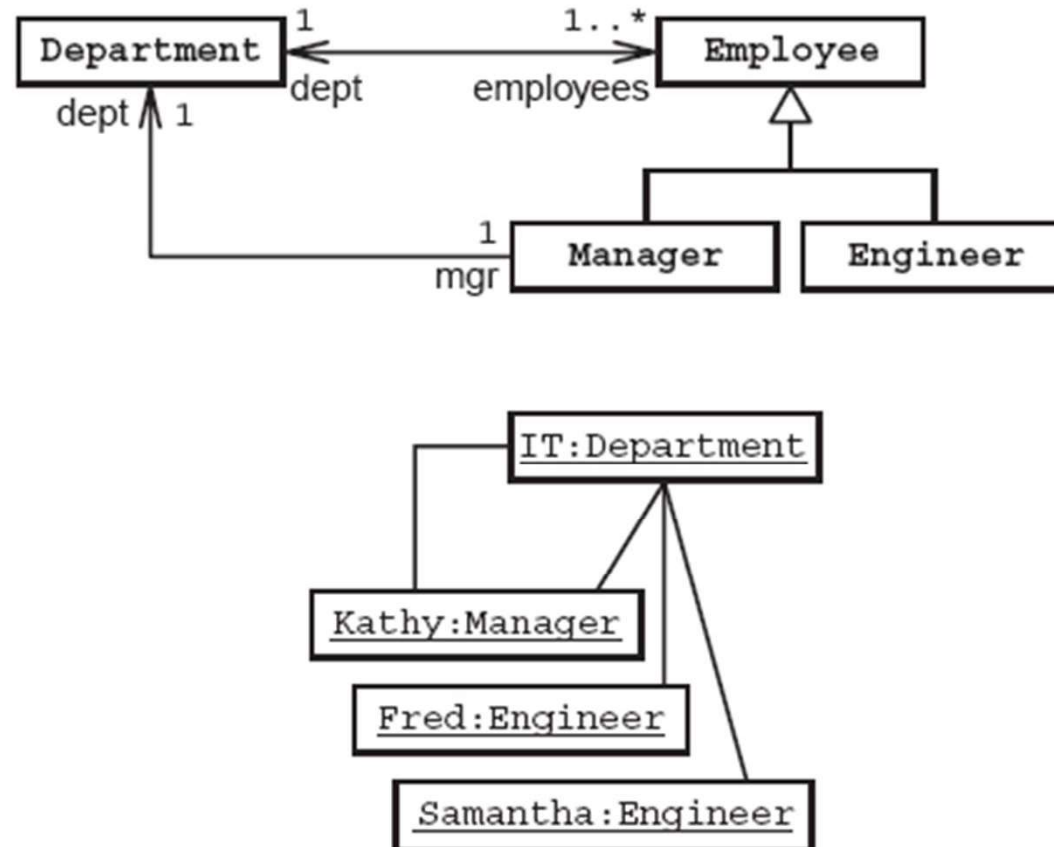
Dimensions of associations include:

- The roles that each class plays
- The multiplicity of each role
 - 1 denotes exactly one
 - 1..* denotes one or more
 - 0..* or * denotes zero or more
- The direction (or navigability) of the association

Object links:

- Are instances of the class association
- Are one-to-one relationships

Class Associations and Object Links: Example



Delegation

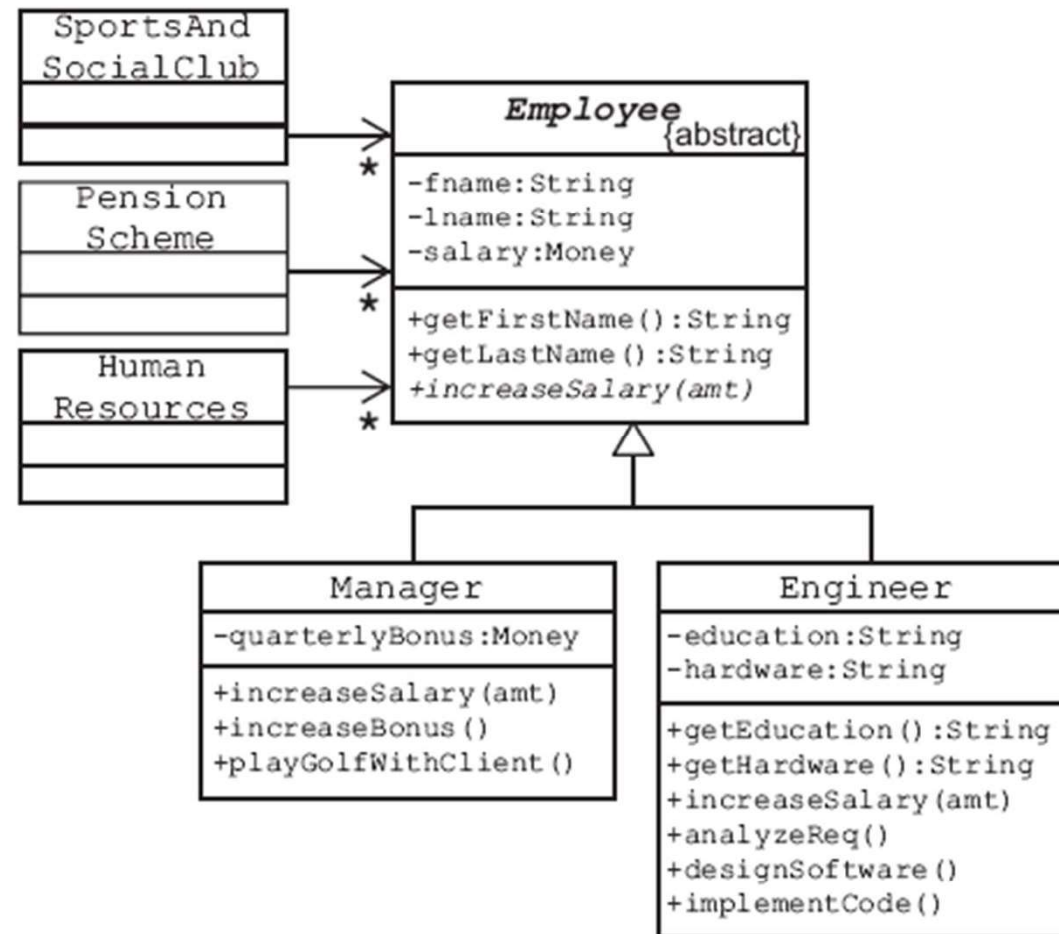
Many computing problems can be easily solved by delegation to a more cohesive component (one or more classes) or method.

- Delegation is similar to how we humans behave.
 - A manager often delegates tasks to an employee with the appropriate skills.
 - You often delegate plumbing problems to a plumber.
 - A car delegates accelerate, brake, and steer messages to its subcomponents, who in turn delegate messages to their subcomponents. This delegation of messages eventually affects the engine, brakes, and wheel direction respectively.
- OO paradigm frequently mimics the real world.

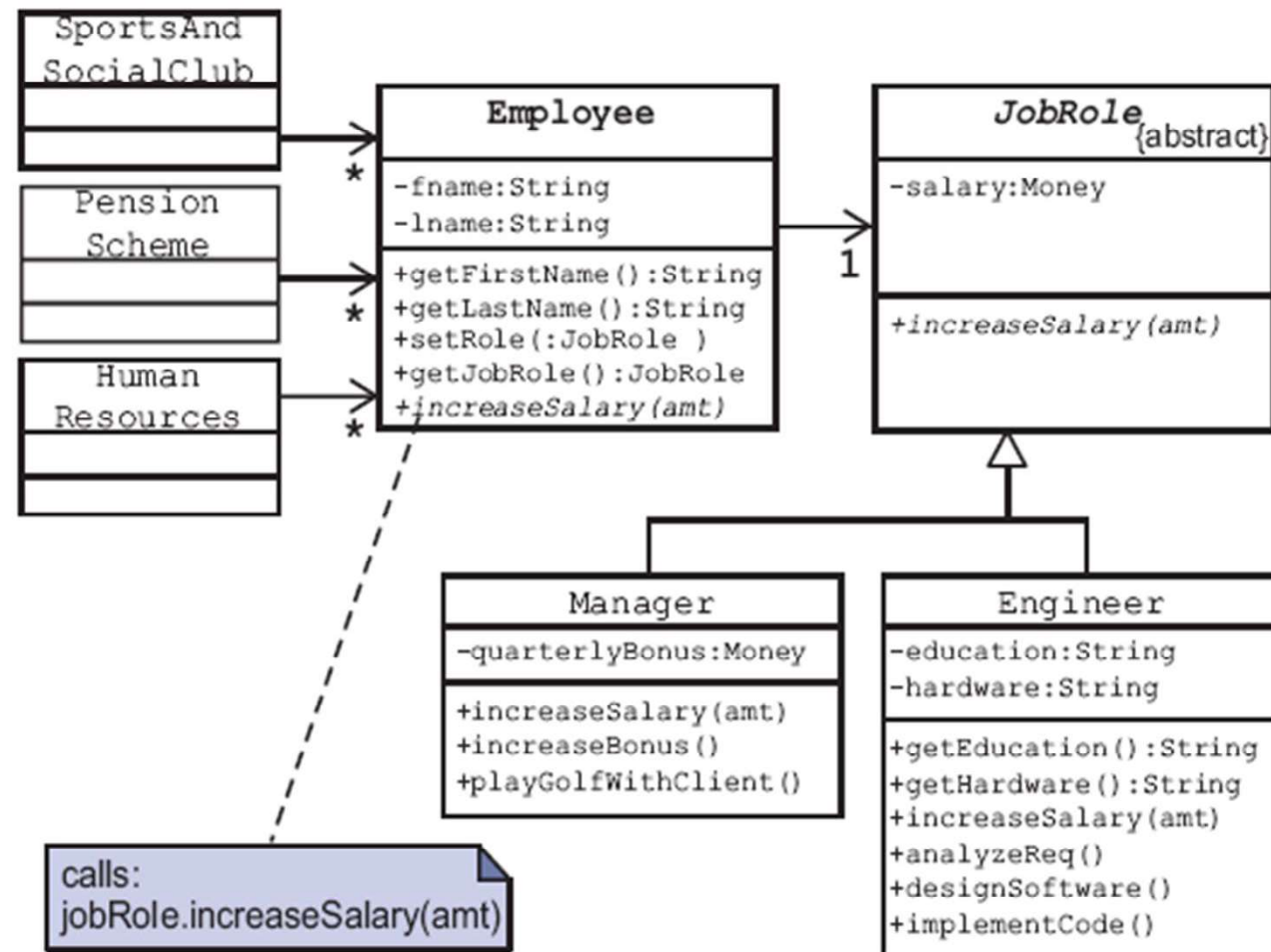
Delegation

- The ways you delegate in OO paradigm include delegating to:
 - A more cohesive linked object
 - A collection of cohesive linked objects
 - A method in a subclass
 - A method in a superclass
 - A method in the same class

Delegation: Example Problem



Delegation: Example Solution



Summary

In this lesson, you should have learned the following:

- Describe the important object-oriented (OO) concepts
- Describe the fundamental OO terminology