

CPSC 5011: Object-Oriented Concepts

Lecture 4: Class Relationships (and UML Class Diagrams)



CPSC 5011: Object-Oriented Concepts

Lecture 4: OOP (Relationships)

Or,

How I ended up
married, 14
years ago today










Caveat, lector...

- UML is a language as much as Java, C++, etc.
- The goal of a diagram is to *simplify* understanding
- Beware mixing logical levels!
 - Eg. Don't mix Class and Object Diagrams
- You do need to know it (or at least know how to figure them out 😊)
- Stephen's rules for UML:
 - If you have to use different arrow symbols, it's too complicated.
 - Sometimes 2 or 3 diagrams are better than 1.
 - I ❤️ Sequence Diagrams (but not relevant here)

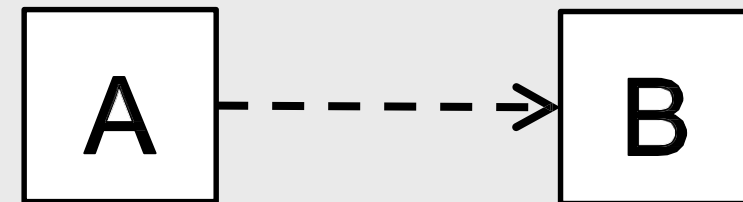
Class Relationships

- UML Connectors: relationships among classes using arrows

	Dependency	"Uses-a"
	Association	
	Direct Association	
	Aggregation	"Has-a"
	Composition	
	Inheritance	"Is-A"
	Interface Implementation (aka realization)	"Implements-A"

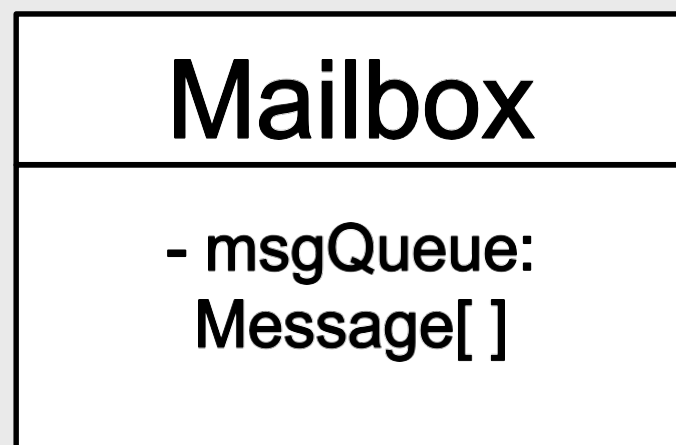
Dependency

- Class **A** uses class **B**
 - i.e. A method of class **Sequence** uses-a **Stack** to verify palindrome
- This is generally a transient relationship
 - i.e. A method of class **A** is passed a parameter of class **B**
 - i.e. A method of class **A** returns a value of class **B**
- No lasting association
 - i.e. Implementation of a method of class **Sequence** that uses-a **Stack** does not affect **Sequence** type definition
- Dependency is asymmetric
 - i.e. The **Stack** class is not aware of the existence of the **Sequence** class.
Therefore, **Stack** objects do not depend on **Sequence** objects
- Loose coupling
 - Minimize the number of dependency relationships
- In UML diagrams,
 - Draw a dashed line with an open arrowhead from class **A** to class **B**

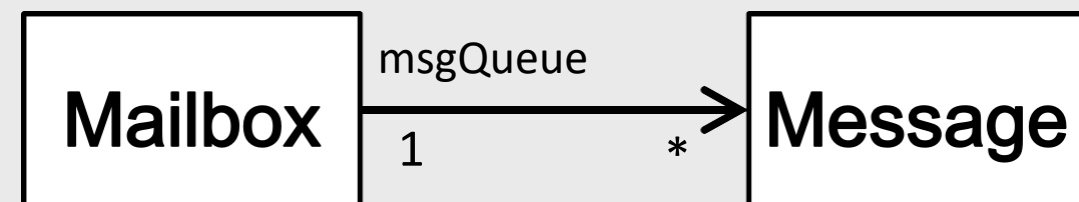


Association

- Class **A** is associated with class **B**
 - A relationship between class **A** and class **B** that lasts as long as class **A** objects and class **B** objects live at runtime
 - Defines the multiplicity between objects
 - one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects
 - In general, class **A** has an *attribute* (field) that is class **B**
- In UML diagrams,
 - Draw a solid line with an open arrowhead from class **A** to class **B**
 - Label the line with the name of the attribute
 - Can also be an aggregation or a composition (more on this later!)
 - Optionally indicate multiplicity

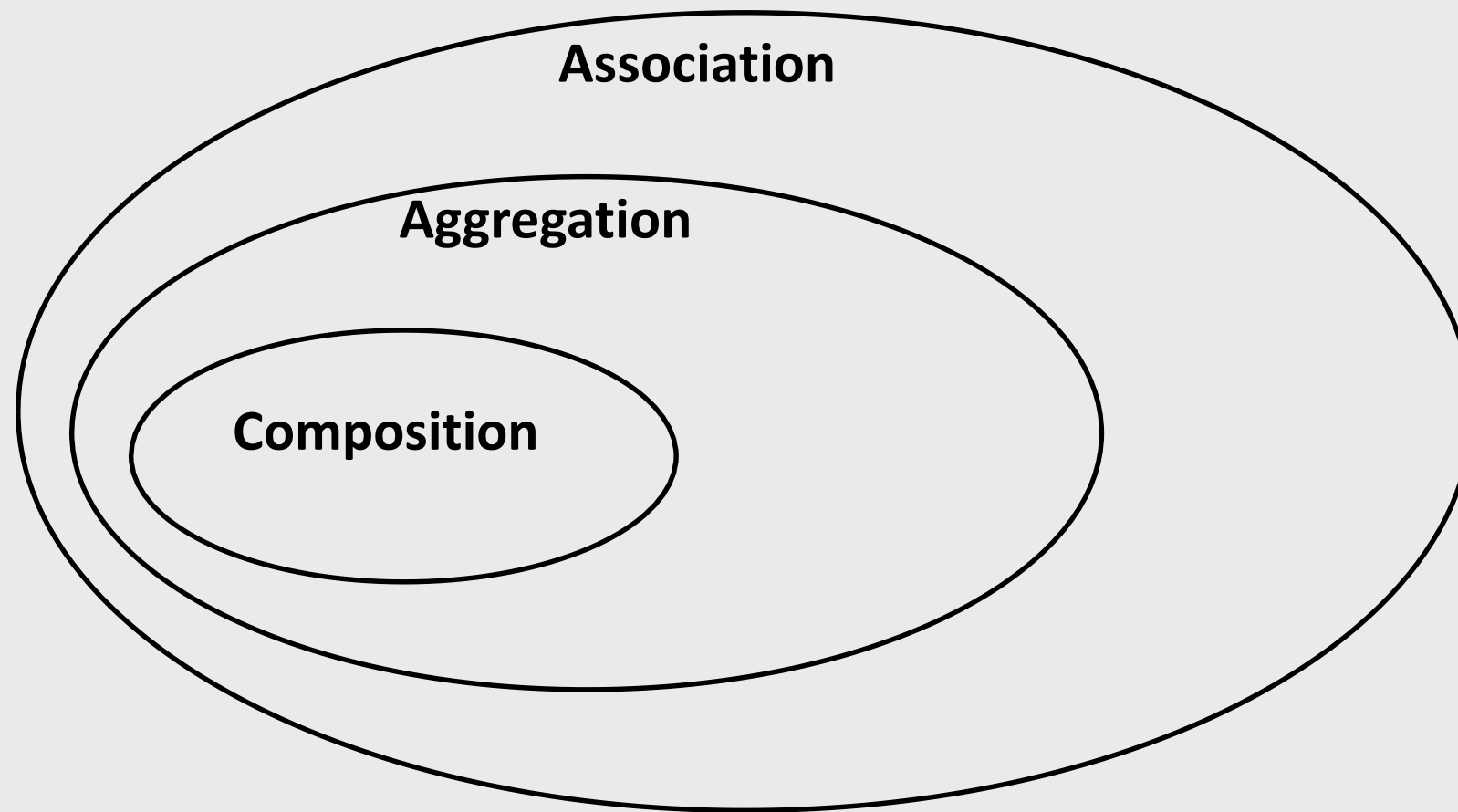


Replace attribute with the association shown below



Association, Aggregation, & Composition

- **Aggregation** is a special form of **Association**
- **Composition** is a special form of **Aggregation**

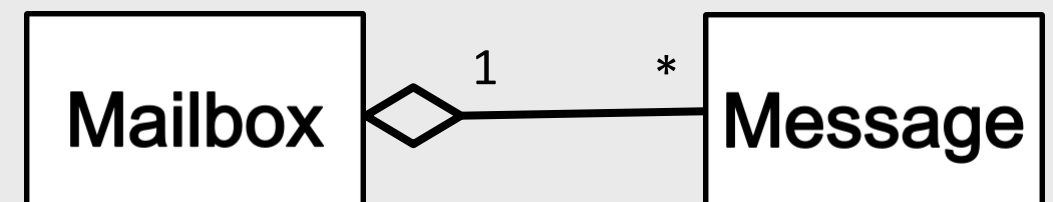


Aggregation

- A special case of (unidirectional) association
 - The “has-a” relationship, i.e. class **A** “aggregates” class **B**
 - Objects of class **A** contain objects of class **B** over a period of time
 - The contained object can have an existence independent of its container
 - Both objects can survive individually; ending one entity will not effect the other
- Example
 - i.e. A **Mailbox** has-a set of **Message(s)**. A **Message** can exist without a **Mailbox**. Therefore, a **Mailbox** aggregates **Message(s)**
- Usually implemented through instance fields
 - i.e. `private List<Message> newMessages;`
 - Simple fields are considered attributes, not aggregation
- Multiplicity
 - i.e. 1:1 – Each **Person** object has-a single **StreetAddress** object
 - i.e. 0:M – Each **Mailbox** object has-a list of multiple **Message** objects

*	zero or more (0:M)
1.. *	one or more (1:M)
0..1	zero or one (0:1)
1	exactly one (1:1)

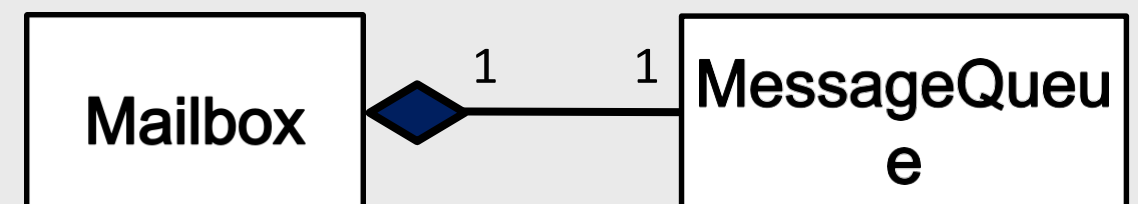
In UML diagrams, draw a solid line with an open diamond head from class **A** to class **B**



Composition

- A special case of aggregation
 - The “has-a” relationship, i.e. class **A** “composes” or “owns” class **B**
 - Represents “part-of” relationship
 - The contained object cannot (logically) have an existence independent of its container
- Type Dependency
 - SubObject is an integral part of type definition
 - OR SubObject is part of the type composition
 - OR Essential component => type dependency
- Example
 - i.e. A **Mailbox** has-a **MessageQueue**. The **MessageQueue** cannot (logically) exist without a **Mailbox**. Therefore, a **Mailbox** composes a **MessageQueue**

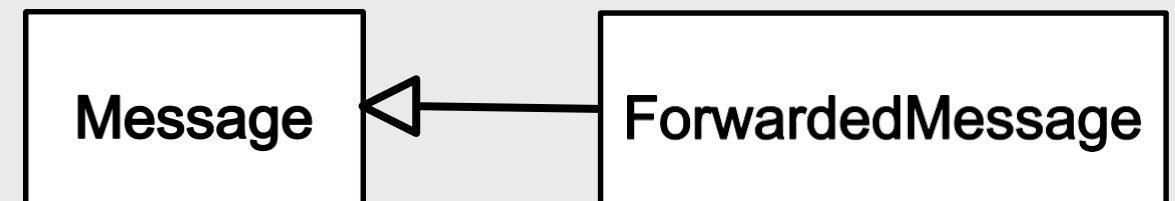
In UML diagrams, draw a solid line with an filled diamond head from class **A** to class **B**



Inheritance

- Class **C** inherits from class **S**
 - The “is-a” relationship, i.e. class **C** “is-a” class **S**
 - All class **C** objects are special cases of class **S** objects
 - Class **S** is the superclass of class **C** ; Class **C** is-a subclass of class **S**
 - An object of class **C** is-an object of class **S**
- Type Dependency
 - Public inheritance: type extension
 - Class hierarchy: Base (parent) and Derived (child) classes
 - Subtype alters or augments inherited behavior
 - Built-in (sub)type checking
- Example
 - i.e. A **ForwardedMessage** object is-a **Message** object
 - i.e. A **Manager** object is-an **Employee** object

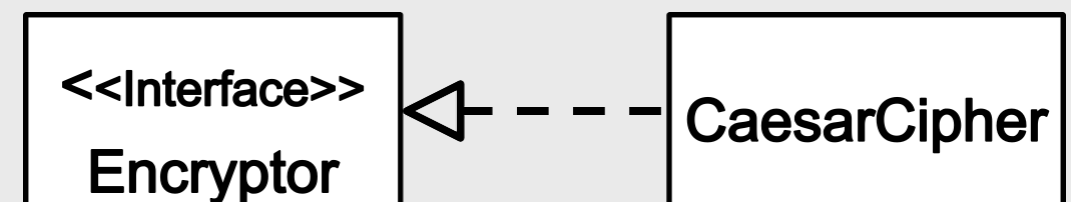
In UML diagrams, draw a solid line with a closed arrow head from class **A** to class **B**



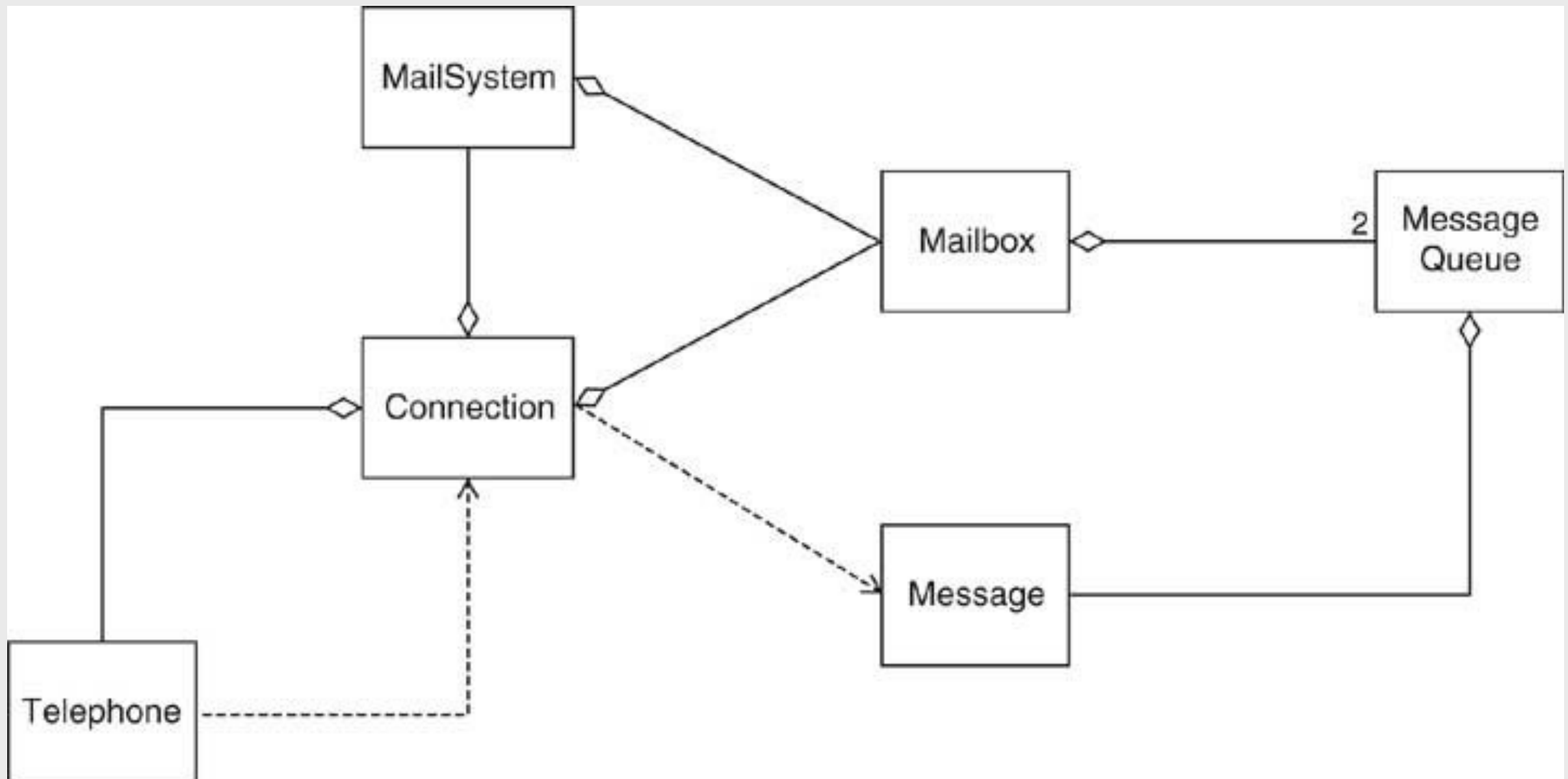
Interface Implementation

- Class **C** inherits from class **I**
 - The “implements-a” relationship, i.e. class **C** “is-a realization of” or “realizes” class **I**, OR class **I** “is realized by” class **C**
 - An interface is similar to a class, but does not provide any implementation
 - A child class must override all methods
 - An interface is like an abstract class in which all methods are abstract
 - Substitutability can occur through the use of interface(s)
 - Subtypes can be formed using interfaces, linking types that have no inheritance relationship whatsoever (more on this later!)
- Example
 - i.e. A **CaesarCipher** object implements-an **Encryptor** object
 - i.e. A **PasswordVault** object implements-a **Vault** object

In UML diagrams, draw a dotted line with a closed arrow head from class **A** to class **B**



Class Diagram – Voicemail System



- **Telephone** depends on **Connection**
- **Connection** depends on **Message**
- A **MailSystem** has **Mailbox(es)**
- A **Mailbox** has 2 **MessageQueue(s)**

- A **MessageQueue** has some number of **Message(s)**
- A **Connection** has a current **Mailbox** ; it also has references to the **MailSystem** and **Telephone** objects that it connects

ICL: UML Class Diagram

- A company has a manufacturing division and a research division.
- The company has a president and two vice presidents. One VP runs manufacturing and the other runs research.
- Each division has employees other than its VP, and it might also have an administrative assistant.
- All manufacturing workers are union members, but none of the researchers. All of the researchers have college degrees, as do some of the manufacturing employees.
- Each employee is paid either by the hour, biweekly, or monthly. Manufacturing workers can get overtime pay, but not researchers. All employees except the president and vice presidents can get bonuses.
- The company manufactures different products, three of which are widgets, gadgets, and gizmos. A gadget can be sold separately, or several can be bundled with a widget. A gizmo can only be sold as part of a gadget.
- The researchers are working on projects to improve the company's existing products or to design new products.

More on Aggregation

Aggregation is a special form of Association that represents a “has-a” relationship and has a unidirectional association i.e. a one way relationship. For example, Department can have Student(s) but vice versa is not possible. In Aggregation, both the entities can survive individually which means ending one entity will not effect the other entity. [Ref: [GeekforGeeks](https://www.geeksforgeeks.org/aggregation-in-uml/)]

- Standard Containers
 - Stack, queue, dictionary, hash table
 - May contain SubObjects, copies or references
 - Number of SubObjects may vary within lifetime
- Object Type not defined by SubObject
 - i.e. Stack well-defined when empty, full or in-between
 - SubObject provides no direct (public interface) functionality for container
- SubObject(s) not necessarily owned by container
 - May be shareable
 - May be passed in, passed out
 - Functions like dequeue() return subObject

Aggregation Example (1)

Department has(-a) Student(s)

- Core Department functionality not defined by Student(s)
- **Department is well-defined without Student**
 - State of student does not *drive* state of department
 - Department may have zero Student(s) and still function as a Department
- **Department may or may not own Student**
 - Department does not hold Student for lifetime
 - Students may be shared, “borrowed”, transferred, ...
 - Department not necessarily responsible for create/destroy
- A given Department may hold varying numbers of Student(s)
 - Cardinality varies over lifetime of Department

```
// Student class
public class Student {
    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

    private String name;
    private int id;
}

// Department class aggregates Student
// class (i.e. Department has Students)
public class Department {
    public Department(String name,
        List<Student> students) {
        this.name = name;
        this.students = students;
    }

    public List<Student> getStudents() {
        return students;
    }

    private String name;
    private List<Student> students;
}
```

Aggregation Example (2)

Team has(-a) Player(s)

- Core Team functionality not defined by Player(s)
- Team is well-defined without Player(s)
- Team may or may not own Player(s)
- A given Team may hold varying numbers of Player(s)

```
// Player class
public class Player {
    public Player(String name) {
        this.name = name;
    }
    private String name;
}

// Team class aggregates Player class
// class (i.e. Team has Players)
public class Team {
    public Team () {
        this.players = new ArrayList<>();
    }
    public void addPlayer(Player p) {
        return players.add(p);
    }
    public void removePlayer(int index) {
        return players.remove(index);
    }
    private List<Player> players;
}
```


More on Composition

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other. It represents part-of relationship. In composition, both the entities are dependent on each other. When there is a composition between two entities, the composed object cannot exist without the other entity. [Ref: [GeekforGeeks](https://www.geeksforgeeks.org/composition-in-java/)]

- **Object Type defined by SubObject(s)**
 - Data membership
 - Typically instantiated upon object construction
 - Cardinality usually fixed within lifetime
 - Correlation between lifetimes
 - SubObject provides functionality
 - **SubObject affects state of Object**
- **SubObject(s) owned**
 - Not usually shareable or transferable
 - May be replaceable
 - Object may be responsible for allocation/deallocation

Composition Example (1)

Plane owns Engine(s)

- Plane not well-defined without Engine(s)
 - Plane not usable without Engine(s)
 - State of Engine(s) affect state of Plane
- Plane owns its Engine(s)
 - Lifetime association between Engine(s) & Plane
 - Engine(s) not shareable with other Plane(s)
 - Plane responsible for creation/destruction of Engine(s)
 - Engine may be replaced by another Engine
- Engine(s) provide needed functionality
- A given Plane has a specific number of Engine(s)
 - Cardinality fixed, typically for lifetime of Plane

```
// Engine class
public class Engine {
    public Engine(String type) {
        this.type = type;
    }
    private String type;
}

// Plane class composes Engine class
// (i.e. Plane owns an Engine)
public class Plane {
    public Plane(Engine e) {
        this.engine = e;
    }
    private final Engine engine;
}
```

Composition Example (2)

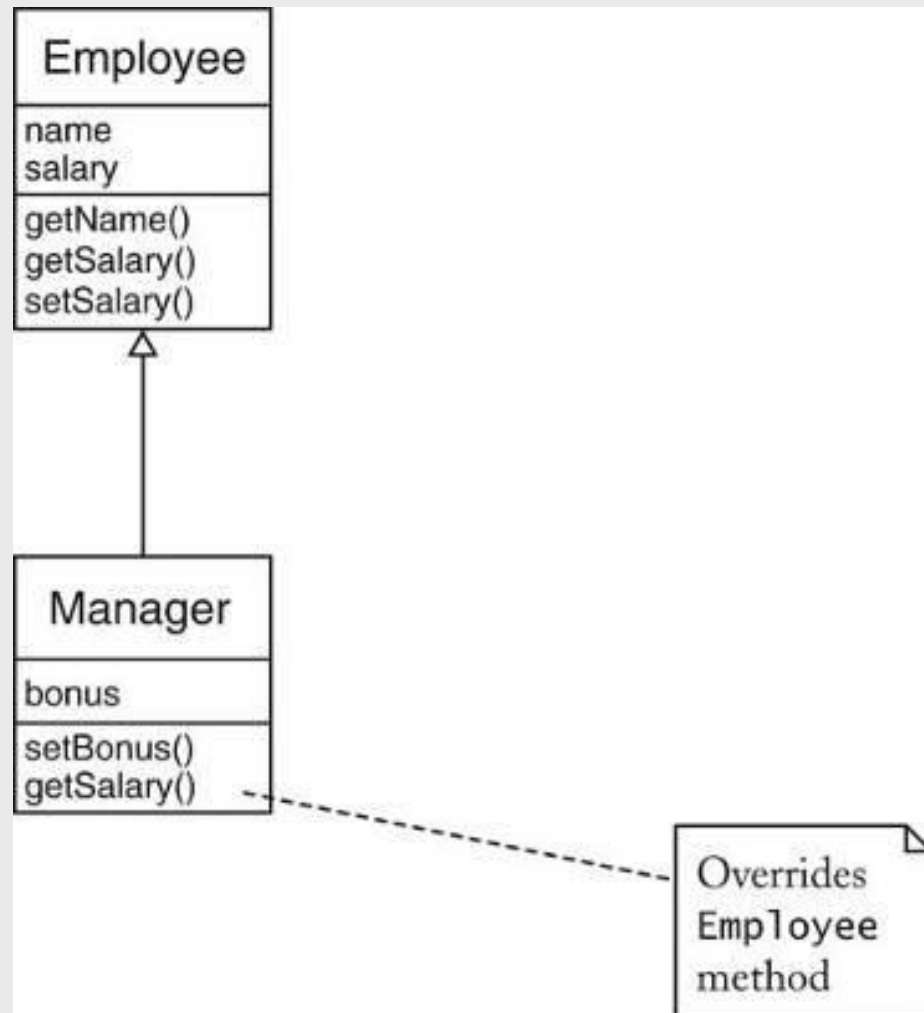
Library owns Book(s)

- Library not well-defined without Book(s)
- Library owns its Book(s)
- Book(s) provide needed functionality
- A given Library has a specific number of Book(s)

```
// Book class
public class Book {
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    private String title;
    private String author;
}

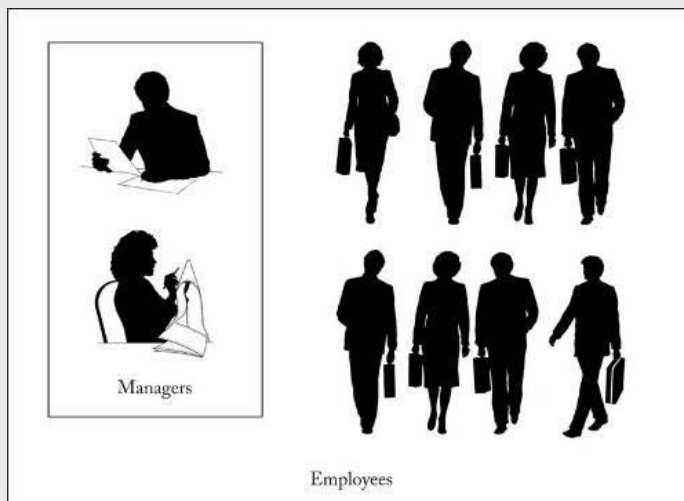
// Library class composes Book class
// (i.e. Library owns Books)
public class Library {
    public Library (List<Book> books) {
        this.books = books;
    }
    public List<Book> getBooks () {
        return books;
    }
    private final List<Book> books;
    // still mutable, but cannot assign
    // books to another reference
}
```

Inheritance Example



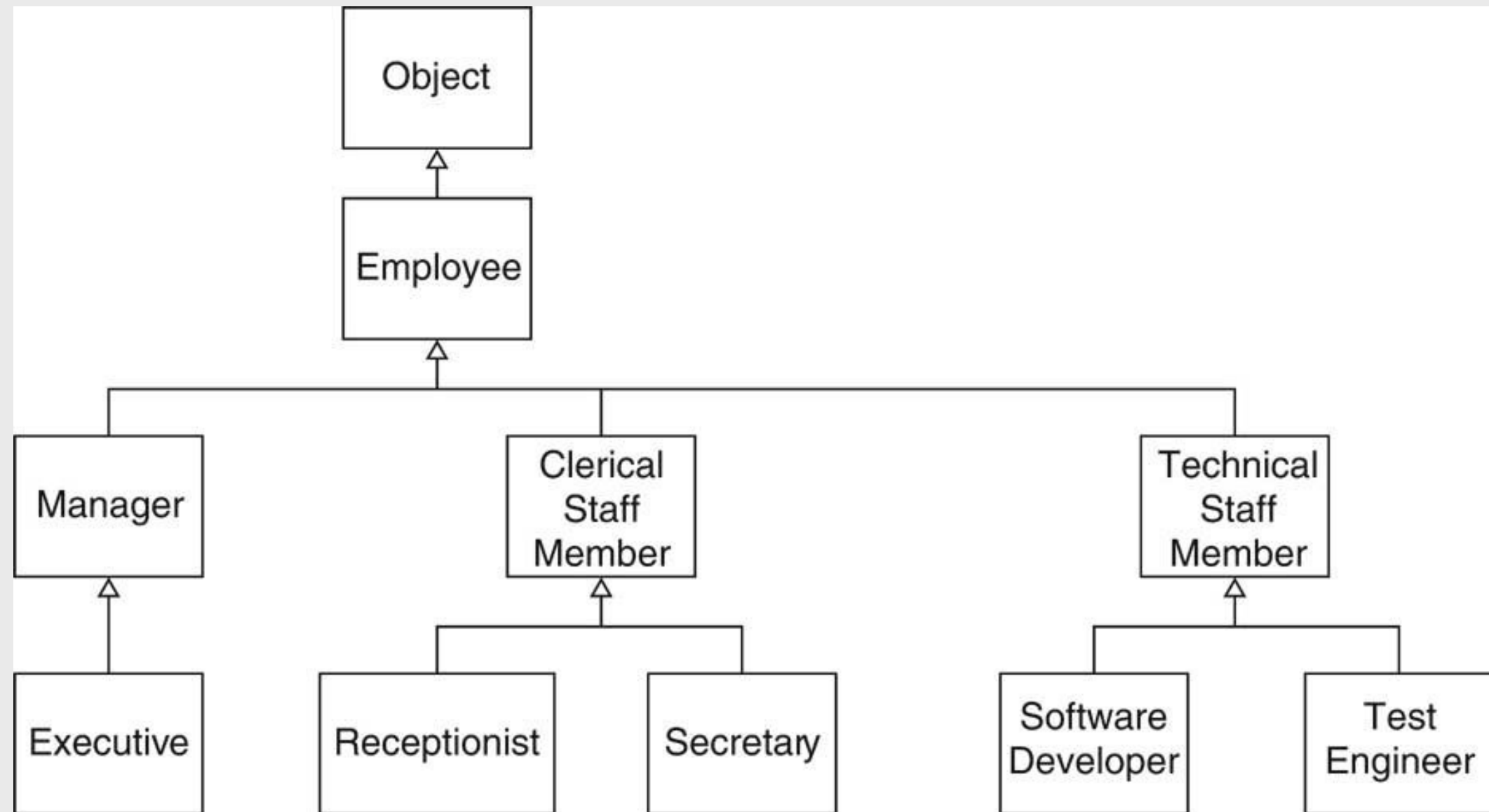
```
// Employee class
public class Employee {
    public Employee(String name) { ... }
    public void setSalary(double salary) { ... }
    public String getName() { ... }
    public double getSalary() { ... }
    private String name;
    private double salary;
}

// Manager class inherits from Employee class
// (Manager is an Employee)
public class Manager extends Employee {
    public Manager(String name) { ... }
    public void setBonus(double bonus) { ... }
    @Override
    public double getSalary() { ... }
    private double bonus; // new field
}
```



- **Employee** is the superclass ; **Manager** is the subclass
- A **Manager** is-an **Employee**
- A **Manager** is-a specialized type of **Employee**
- The set of **Manager** objects is a subset of the set of **Employee** objects

Class (Inheritance) Hierarchies



- Object-oriented programming uses class hierarchies
 - The most general superclass is at the root of a tree (i.e. **Object** in Java)
 - More specific subclasses are children (i.e. **Manager** is-an **Employee**)
 - The most specific subclasses are leaves (i.e. **Executive** is-a **Manager**; in addition, **Executive** is-an **Employee** and **Executive** is-an **Object**)

Invoking Superclass Methods

- Can't access private fields of superclass

```
public class Manager extends Employee {  
    public double getSalary() {  
        return salary + bonus;    // ERROR: private field  
    }  
    ...  
}
```

- Be careful when calling superclass method

```
public double getSalary() {  
    return getSalary() + bonus; // ERROR: recursive call  
}
```

- Use **super** keyword

```
public double getSalary() {  
    return super.getSalary() + bonus;  
}
```

- **super** is not a reference
- **super** turns off polymorphic call mechanism (more on this later!)

Invoking Superclass Constructor

- Use **super** keyword in subclass constructor:

```
public Manager(String aName) {  
    super(aName); // calls superclass constructor  
    bonus = 0;  
}
```

- Call to **super** must be first statement in subclass constructor
- If subclass constructor doesn't call **super**, superclass must have constructor without parameters

Preconditions & Postconditions

- **Precondition** of redefined method at most as strong

```
public class Employee {  
    /**  
        Sets the employee salary to a given value.  
        @param aSalary the new salary  
        @precondition aSalary > 0  
    */  
    public void setSalary(double aSalary) { ... }  
}
```

- Can we redefine `Manager.setSalary` with precondition `salary > 100000`?
=> No; could be defeated by `Employee`.

```
i.e. Manager m = new Manager();  
     Employee e = m;  
     e.setSalary(50000);
```

- **Postcondition** of redefined method at least as strong
 - Example:
 - `Employee.setSalary` promises not to decrease salary
 - `Manager.setSalary` must fulfill postcondition
 - Redefined method cannot be more private. Common error: omitting `public` when redefining.
Redefined method cannot throw more checked exceptions

Structural Design Details

- **Cardinality** -- How many SubObjects?
 - Inheritance: Fixed, 1:1
 - Composition: 1:M
 - Aggregation: Variable
- **Ownership**
 - Inheritance: Child owns parent component; may NOT be released
 - Composition: Composing object may stub out/replace SubObject
 - Aggregation: None (usually)
- **Lifetime**
 - Inheritance: 1:1
 - Composition: Variable by design
 - Aggregation: Object lifetime does not affect subObject's lifetime (and vice versa)
- **Association**
 - Inheritance: Permanent
 - Composition: Stable, possibly transient
 - Aggregation: Temporary

Type Extension

- Pure form of inheritance; inheritance of interface
 - Type extension (**PUBLIC**)
 - Child type **is-a** (extension of) parent type
 - Child type retains, supports and, possibly extends parent interface
- Unambiguously supports **is-a** relationship
 - Parent type not compromised
- Child class retains all properties of parent class
- Child class adds attributes and/or methods
- Behavior (& data) associated with child class is an extension of parent class
 - Strictly larger set of functionality
- Example
 - A **BiAthlete** is-a **Runner** *add biking*
 - A **TriAthlete** is-a **BiAthlete** *add swimming*

Code Reuse

- Impure form of inheritance; inheritance of implementation
 - Specialization, subtyping, contraction
 - Code reuse **(PRIVATE)**
 - Already defined, debugged, tested
 - Child class inherits and uses parent code
- Does NOT completely support **is-a** relationship
- Child class inherits all from parent class BUT
 - Redefines or limits properties of parent class
 - Overrides or redefines behavior inherited from parent
 - Suppresses all or part of parent interface
- Child class more restricted form of parent class
- Example:
 - A **PriorityQ** is-a (ordered) **Queue**
 - A **BST** is-a **BinaryTree** with special properties

Advantages

- **Code reuse**
 - Reduce development cost (& time)
 - Usually better maintenance
 - Less cut & paste programming
 - Parent class presumed stable
 - already designed, implemented, debugged and tested
- **Type extension**
 - Application Programmer familiar with parent
 - Substitutability
 - Polymorphism – run-time selection of functions
 - New type can be added without breaking application code

Disadvantages

- Increased Coupling
 - Child tightly coupled to parent
- Decreased Cohesion
 - Type definition spread across inheritance hierarchy
- Fixed Relationship
- Overhead
 - Child absorbs overhead of parent component even if not used
- Maintenance
 - Cost highly dependent on design

Applications of Inheritance

1) Specialization (subtyping)

- Child provides or defines special behavior
 - Distinguish child from parent by detail(s)
 - Budd: Each child class overrides a method inherited from the parent in order to specialize the class in some way. Most common form of inheritance.
- A is-a B => A can be used in place of B
- **Example** : A priorityQ is-a Queue
 - Is-a relation retained
 - priorityQ can be used as a Queue object
 - Interface consistent
 - enqueue() implementation modified to order entries by priority

Applications of Inheritance

2) Specification

- Abstract Class (parent) defines base of hierarchy
 - Incomplete definition
 - Not instantiable – no objects constructed from Abstract class
- Parent defines interface (methods)
 - Implementation deferred to child class
 - Core set of required behaviors defined
 - Common interface
- Child class defines (specifies) behavior outlined by parent
 - Budd: If the parent class is abstract, we often say that it is providing a specification for the child class, and therefore it is specification inheritance (a variety of specialization inheritance).
 - Child class
 - not refinement of existing usable type
 - realization of incomplete abstract specification
- **Example** : A Car is-a Vehicle
 - Vehicle's definition insures that all derivations (car, plane, boat) move

3) Construction

- Budd: If the parent class is used as a source for behavior, but the child class has no is-a relationship to the parent, then we say the child class is using inheritance for construction.
- Private inheritance
 - suppress inherited interface
 - No intent to sustain relation
 - Code reuse
- No is-a relation
 - No substitutability – violates principle of substitutability
- A Set derived from a PriorityQueue
 - Application programmer cannot use Set as PriorityQueue
 - Question: Why not use composition instead?

Applications of Inheritance

4) Generalization

- Budd: If a child class generalizes or extends the parent class by providing more functionality, but does not override any method, we call it inheritance for generalization. The child class doesn't change anything inherited from the parent, it simply adds new features.
- Goal is to create more general class
 - Extend use by application programmer
 - Expand platform for subsequent derivation
- Build on existing class that cannot be modified
 - Code reuse
 - Inverted class hierarchy
 - No proper is-a
- Parent interface may be compromised
- Subclass modifies or extends properties of parent class
 - Impure expansion
 - Modifies at least one inherited method
- **Example** : A Player is-a Warrior

5) Extension

- Budd: If a child class generalizes or extends the parent class by providing more functionality, but does not override any method, we call it inheritance for generalization. The child class doesn't change anything inherited from the parent, it simply adds new features.
- **Is-a relation supported**
- Parent interface extended
 - Add new abilities by adding new methods
- Does NOT override parent class methods
- **Example** : A TriAthlete is-a BiAthlete
 - Runs and bikes but ALSO swims

Applications of Inheritance

~~6) Limitation~~

- Budd: If a child class overrides a method inherited from the parent in a way that makes it unusable (for example, issues an error message), then we call it inheritance for limitation.
 - i.e. You have an existing List data type that allows items to be inserted at either end, and you override methods allowing insertion at one end in order to create a Stack.
 - Generally not a good idea, since it breaks the idea of substitution. But again, it is sometimes found in practice.
- Special case of subclassing for specialization
- Often used when building on existing classes
 - Code reuse
 - **Violates is-a**
- Behavior of subclass
 - smaller than parent (suppressed interface)
 - more restricted than parent (conditional use)
- **Example** : A Stack is derived from a Vector (or List)
 - Restrict access to only one end

Applications of Inheritance

~~7) Variance~~

- Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child
 - similar implementation
 - no hierarchical relationship
- Build on existing class that cannot be modified
 - Code reuse
 - Forced relation is arbitrary
- Better solution? Abstract out common parts to new parent class, and use subclassing for specialization
- **Example** : A Knight is-a Ninja or A Ninja is-a Knight
 - Should be a Ninja is-a Warrior

8) Combination

- Multiple inheritance
- Child class derived from two or more parents
 - Student-Employee is-a Student AND is-an Employee
- Complex relations
 - One parent could be used for extension
 - Another parent could be used for construction
 - => Mixed intent of design
- Potential problems
 - Redundancy
 - Ambiguity

Summary of Forms of Inheritance

- 1) **Specialization**. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- 2) **Specification**. The parent class defines behavior that is implemented in the child class but not in the parent class.
- 3) **Construction**. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- ~~4) **Generalization**~~. The child class modifies or overrides some of the methods of the parent class.
- 5) **Extension**. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- ~~6) **Limitation**~~. The child class restricts the use of some of the behavior inherited from the parent class.
- ~~7) **Variance**~~. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- 8) **Combination**. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

Problem of Defining Types

Consider how we might define a Stack ADT:

```
interface Stack {  
    public void push (Object value);  
    public Object top ();  
    public void pop ();  
}
```

Notice how the interface itself says nothing about the LIFO property, which is the key defining feature of a stack. Is the following a stack?

```
class NonStack implements Stack {  
    public void push (Object value) { v = value; }  
    public Object top () { return v; }  
    public void pop () { v = null; }  
    private Object v = null;  
}
```

Note: A subtype preserves the meaning (purpose, or intent) of the parent. Problem? Meaning is extremely difficult to define. Think about how to define the LIFO characteristics of the stack. The NonStack is a subclass, but not a subtype.

Substitution & Relationship Issues

Substitution Paradox (for strongly typed OO languages)

- Substitution is permitted, based on subclasses. A variable declared as the parent type is allowed to hold a value derived from a child type. Yet from a semantic point of view, substitution only makes sense if the expression value is a subtype of the target variable.
- If substitution only makes sense for subtypes and not for all subclasses, why do programming languages base the validity of assignment on subclasses?

The Undecidability of the Subtype Relationship

- It is trivial to determine if one class is a subclass of another. It is extremely difficult to define meaning (think of the Stack ADT), and even if you can, it is almost always impossible to determine if one class preserves the meaning of another.
- One of the classic corollaries of the halting problem is that there is no procedure that can determine, in general, if two programs have equivalent behavior.

Is this a problem?

- What does it take to create a subclass that is not a subtype?
 - The new class must override at least one method from the parent
 - It must preserve the type signatures
 - But it must violate some important property of the parent
- Is this common? Not likely. But it shows you where to look for problem areas.

Subclass vs Subtype

- **Ideal:** **type extension**
 - Substitutability desired in application
 - Extensibility needed
- **Practical motive:** **code reuse**
 - Significant portion of parent code reused by child
- **Consider**
 - Applications' need for extensibility
 - Heterogeneous collections
 - Stability of parent class
 - has-a as an alternative
 - ability to avoid overhead when desired
 - Document design choice(s)
- **Subclass vs subtype**
 - To say that A is a **subclass** of B merely asserts that A is formed using inheritance
 - To say that A is a **subtype** of B asserts that A preserves the meaning of all the operations in B
 - It is possible to form subclasses that are not subtypes; and (in some languages at least) form subtypes that are not subclasses

The Liskov Substitution Principle (LSP)

Given a type T with a subtype S defined via inheritance, any object of subtype S can serve in place of an object of type T

- Named after Barbara Liskov
 - MIT computer science professor
 - Pioneer in object-oriented programming
- In Java, you should be able to substitute a superclass object by a subclass object

- **Example:**

```
Employee e = new Employee("John Doe");  
System.out.println("salary: " + e.getSalary());
```

```
Employee e = new Manager("Mary Jane");  
System.out.println("salary: " + e.getSalary());
```

- The type of variable **e** is **Employee**. The type of the object is **Manager**.
 - How does Java know which `e.getSalary()` method to invoke at runtime?
 - The JVM uses the type of the object, not the type of the variable => Polymorphism

Question : How is LSP related to the principle of programming to the interface?

Answer : Type type of the variable e is the superclass Employee instead of the more specific subclass type Manager.