Classes

- **Class**
  - Heart of object-oriented systems
  - Types of objects needed in your system to accomplish the use cases
    - Abstraction
    - Encapsulation: data + operations on the data in one place
      - Increase reusability

- **Notation**

Class Diagram

- **Shows:**
  - Classes of a system
  - Interaction between classes
  - Operations and attributes of the classes

- **Used to:**
  - Explore domain concepts in the form of a domain model
  - Analyze requirements in the form of a conceptual model
  - Depict detailed design of object-oriented software
Classes: Visibility (cont.)

- Visibility of attributes and operations
  - Protected
    - Specified using hash (#) symbol
    - Only accessible by methods that are part of your class and also by methods that are declared on any child class
  - Package
    - Specified using tilde (~) symbol
    - Only accessible by methods of classes within same package, without regard to the inheritance relationship
  - Private
    - Specified using minus (-) symbol
    - Only the class that contains the private element can see or work with the data stored in a private attribute or make a call to a private operation.

Classes: Visibility (cont.)

- Visibility of attributes and operations
  - Public

Class State: Attributes

- Attributes represent the state of an object
- In class diagram, represented as:
  - Inline attributes, or
  - Association with another class
- Minimum requirements: visibility, name, and type
**Multiplicity**

- Specify mapping relationship between class instances
  - Additional properties can be added.
  - e.g., ordered, unique, readOnly
  - By default, all attributes with multiplicity are unique.
  - Authors [1..5] / Trackbacks [*] / Comments [*]

**Class Behavior: Operations**

- Operations describe *what* a class can do, but not necessarily how it is going to be.
- Operation signature contains at minimum:
  - Visibility property
  - Name and parameters
  - Return type

**Static Part of a Class**

- Static attributes and operations?
  - Associated with classes, not individual objects
  - *Same copy of static attributes and operations are shared by all class instances (i.e., objects).*
  - Represented by underlining attributes or operations
Relationships between Classes

- **Dependency**
  - Objects of one class work briefly with objects of another class
  - Weakest direct relationship between classes

  ![Dependency Diagram](dependency diagram)

Relationships between Classes (cont.)

- **Association**
  - One class use objects of another class consistently.
  - i.e., a class actually contains a reference to an object or objects of the other class as an attribute.

  ```java
  public class BlogAccount {
      // Attribute introduced thanks to the association with the BlogEntry class
      private BlogEntry[] entries;
      // ... Other Attributes and Methods declared here ...
  }
  public class BlogEntry {
      // Attribute introduced thanks to the association with the Blog class
      private BlogAccount blog;
      // ... Other Attributes and Methods declared here ...
  }
  ```

Relationships between Classes (cont.)

- **Navigability**
  - Describe which class contains the attribute
  - Cross for no navigability

  ```java
  public class BlogAccount {
      // Attribute introduced thanks to the association with the BlogEntry class
      private BlogEntry[] entries;
      // ... Other Attributes and Methods declared here ...
  }
  public class BlogEntry {
      // The blog attribute has been removed as it is not necessary for the
      // BlogEntry to know about the BlogAccount that it belongs to.
      // ... Other Attributes and Methods declared here ...
  }
  ```

Relationships between Classes (cont.)

- **Association class**
  - Class for attributes related to all associated classes
  - No hard rules in implementation

  ```java
  public class BlogAccount {
      private String name;
      private Category[] categories;
      private BlogEntry[] entries;
  }
  public class Category {
      private String name;
  }
  public class BlogEntry {
      private String name;
      private Category[] categories
  }
  ```
Relationships between Classes (cont.)

- **Aggregation**
  - Stronger version of association
  - Has-a relationship
  - Indicate that a class owns but may share objects of another class

![Aggregation Diagram](image)

- **Composition**
  - Stronger relationship than aggregation
  - Part-a relationship
  - Parts would not be shared with other system parts
  - Parts will be deleted if the whole object is deleted.

![Composition Diagram](image)

Relationships between Classes (cont.)

- **Generalization (Inheritance)**
  - Describe a class that is a type of another class
  - Is-a relationship
  - Implementation is reused
  - Tight coupling between child and parent class
  - Strongest form of class relationship

![Generalization Diagram](image)

Relationships between Classes (cont.)

- **Multiple inheritance (or multiple generalization)**
  - Inherit from multiple parents
  - Not considered as a good practice in general
  - Complications
    - Entries are in both parents which one to inherit?
    - What if both parents have same operation?

![Multiple Inheritance Diagram](image)
Constraints

- Restrict the ways in which a class can operate.
  - Use OCL (Object Constraint Language) in curly brackets or in a separate note
- Example constraints
  - Class invariants
  - Method pre-conditions
  - Method post-conditions
  - Dependency between attributes' values

Abstract Classes

- Parent class in generalization may not implement all behavior, but leave to sub classes.
- Contain abstract operations
- Represented by writing method signatures in italics
- If any parts of a class are abstract, the class need to be declared as abstract, too.

Example

```java
public abstract class Store {
    public abstract void store(Article[] articles);
    public abstract Article[] retrieve();
}
```

Interfaces

- Collection of operations that have no corresponding method implementations
- Similar to an abstract class that contains only abstract methods – no state!
- Can avoid problems associated with multiple inheritance
- Help achieve loosely-coupled classes by separating behavior from implementation
- Represented as stereotyped class

```java
public interface EmailSystem {
    public void send(Message message);
}
Interfaces (cont.)

- Interfaces cannot be instantiated by itself, like an abstract class.
- But, they are real\emph{ized} by a class.

```
public interface EmailSystem {
    public void send(String message);
}
```

```
public class SMTPMailSystem implements EmailSystem {
    public void send(String message) {
        // Implement the interactions with an SMTP server
    }
    // ... Implementations of the other operations ...
}
```

It is a good practice to de-couple dependencies between classes using interfaces!

Templates

- Parameterized class

```
List<Things> {
    + elements: E
    + add(Things element)
    - remove(Things element)
    - size() : int
    - contains(Things element) : boolean
}
```

- Needs to be bound to actual type by
  - Sub-classing
  - Runtime parameter binding
  - Type of parameters will be known as it is constructed into an object

Sources: Learning UML 2.0 (by R. Miles and K. Hamilton)
Modeling Object Instances

- Runtime logical view of a system model
  - Describe how all of the different types of objects in a system interact with each other.

- Object instances
  - Notation (underlined)

![Image of object instances](image)

Links and Constraints

- Communication line between objects instantiated from associated classes.
  - Can have a label that describes the link purpose
- Constraints in class diagram must be kept when adding links between objects

![Image of links and constraints](image)

Links and Constraints (cont.)

- OCL to describe constraints for object instantiation
  - Objects and links must abide by the rules set by OCL statements.

![Image of links and constraints](image)

Binding Class Templates

- Object diagram is ideal for modeling runtime parameter binding.

![Image of binding class templates](image)