UML Class Diagrams

A class diagram depicts the structure of an object-oriented system by showing the classes (and interfaces) in that system and the relationships between the classes (and interfaces).

Depicting a Class

In a UML diagram, each class or interface is represented by a rectangle, which is then divided into 3 sections: one for the name of the class or interface, one for the instance variables, and one for the methods.

Class names are written exactly as they appear in the program, including any capitalization. Abstract classes italicize their class names in a UML diagram to differentiate it from a regular class. For similar reasons, interfaces have the phrase "<<interface>>" written above the class name. For example:

- public class MyClass { … } → MyClass
- public abstract class MyClass { … } → *MyClass*
- public interface MyInterface { … } → <<interface>> MyInterface

Instance variables methods are listed one per line, beginning with one of the following symbols to represent its visibility:

- + : Denotes that the given instance variable or method s public
- —: Denotes that the given instance variable or method is private
- # : Denotes that the given instance variable or method is protected

After the visibility modifier, the name of the instance variable is listed, followed by a ":" and the type of that variable. Final variables also list their assigned value. For example:

- public String code → + code : String
- private double secretNum → —secretNum : double
- protected boolean isTrue → # isTrue : boolean
- public static final int SIX = 6 → + SIX : int = 6

Methods are also listed one per line, beginning with a symbol to represent its visibility. The name of the method is then given, along with the parameter list. Each parameter follows the pattern "pName : pType," similar to the pattern described above for instance variables (except that no visibility symbol is included). The parameters are then followed by a ":" and the type that is returned by the function. For example:

- public static void main(String[] args) → + main(args : String[]) : void
- private double calculate(int input, double divider) → —calculate(input: int, divider: double) : double
- protected String helperMethod(String name) → # helperMethod(name: String) : String

Example of a UML diagram for a single class (based on the code on pg 186 of the textbook):

```
public class Coin {
        private final int HEADS = 0;
        private int face;

        public Coin() { … }
        public void flip() { … }
        public void flip (int numTimes) { … }
        public boolean isHeads() { … }
        public String toString() { … }
}
```

Coin
—HEADS : int = 0 —face : int
+ Coin() + flip() : void + flip(numTimes : int) : void + isHeads() : boolean + toString() : String

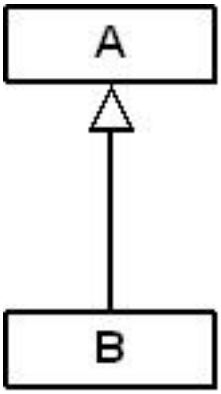
Relationships

The most common relationships are: generalization, realization, association, and dependency.

Generalization Relationship

A generalization is a relationship between a general thing (called the superclass or parent class) and a more specific kind of that thing (called the subclass or child class). Generalization is sometimes called an “is-a” relationship and is established through the process of inheritance.

In a class diagram, generalization relationship is rendered as a solid directed line with a large open arrowhead pointing to the parent class.



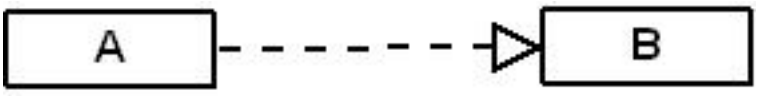
Use generalizations when you want to show parent/child relationships. The above class diagram reflects the relationships present in the following Java code fragment:

```
public class B extends A {
	…
}
```

Realization Relationship

A realization is a relationship between two things where one thing (an interface) specifies a contract that another thing (a class) guarantees to carry out by implementing the operations specified in that contract.

In a class diagram, realization relationship is rendered as a dashed directed line with an open arrowhead pointing to the interface.



Use realizations to show the relationship between a class and an interface. The above class diagram reflects the relationships present in the following Java code fragment. In this case, class A implements the contract (operations) specified in the interface B.

```
public class A implements B {
	…
}
```

Association Relationship

Some objects are made up of other objects. Association specifies a “has-a” or “whole/part” relationship between two classes. In an association relationship, an object of the whole class has objects of part class as instance data.

In a class diagram, an association relationship is rendered as a directed solid line.

If there is an association relationship between class A and class B, i.e., class A “has-a” class B, then the following statements hold true:

1. Class B is used as the type of one or more fields (instance or class variables) in class A.

Aggregation and composition are two variants (or special forms) of the association relationship. In a class diagram, aggregation relationship is rendered as a solid line with an open diamond near the whole class.

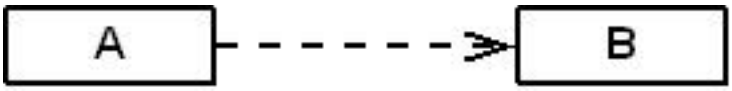
Composition is rendered as a solid line with a filled diamond near the whole class.

For the purposes of this course, we will treat association, aggregation, and composition relationships as equal. We will use only the association relation and ignore its special forms (aggregation and composition) in this course, but be aware that your textbook does use these diamonds in its diagrams.

Dependency Relationship

Dependency indicates a “uses” relationship between two classes.

In a class diagram, a dependency relationship is rendered as a dashed directed line.



If a class A “uses” class B, then one or more of the following statements generally hold true:

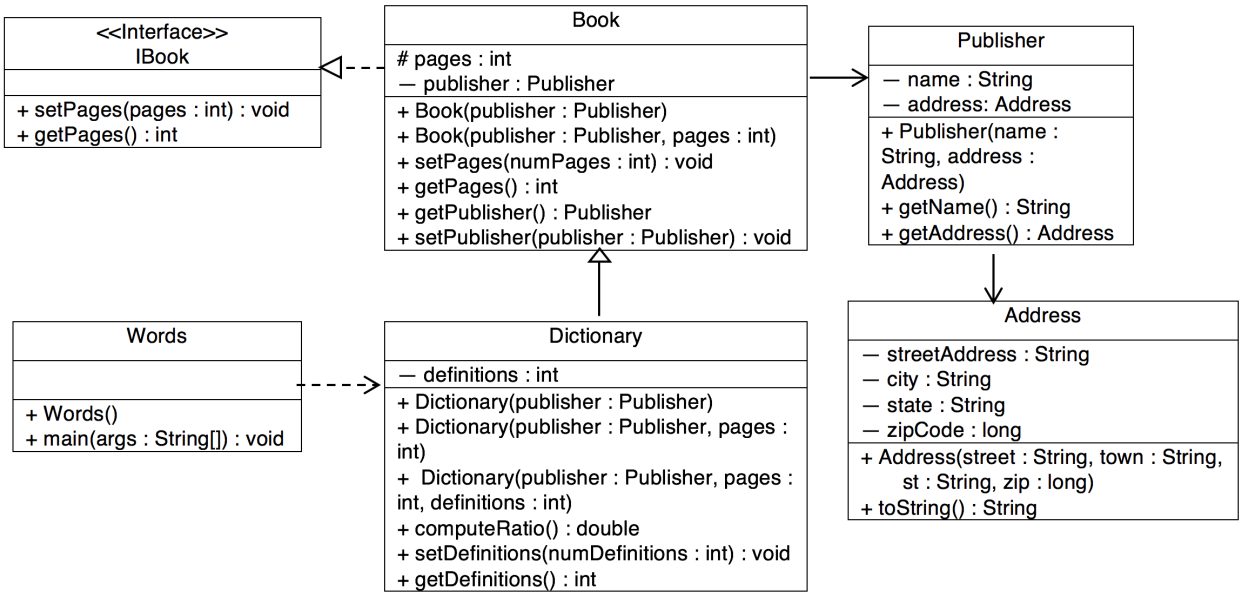
1. Class B is used as the type of a local variable in one or more methods of class A.
2. Class B is used as the type of parameter for one or more methods of class A.
3. Class B is used as the return type for one or more methods of class A.
4. One or more methods of class A invoke one or more methods of class B.

Example: Consider the following program with five classes and one interface. The corresponding class diagram is shown on the last page.

```
public interface IBook {
	public abstract void setPages(int pages);
	public abstract int getPages();
}

public class Book implements IBook {
	protected int pages = 1500;
	private Publisher publisher;

	public Book(Publisher publisher) {
		this.publisher = publisher;
	}

	public Book(Publisher publisher, int pages) {
		this.publisher = publisher;
		this.pages = pages;
	}

	public void setPages(int numPages) {
		pages = numPages;
	}

	public int getPages() {
		return pages;
	}

	public Publisher getPublisher() {
		return publisher;
	}

	public void setPublisher(Publisher publisher) {
		this.publisher = publisher;
	}
}

public class Dictionary extends Book {
	private int definitions = 52500;

	public Dictionary(Publisher publisher) {
		super(publisher);
	}

	public Dictionary(Publisher publisher, int pages) {
		super(publisher, pages);
	}

	public Dictionary(Publisher publisher, int pages, int definitions) {
		super(publisher, pages);
		this.definitions = definitions;
	}

	public double computeRatio() {
		return definitions/pages;
	}

	public void setDefinitions(int numDefinitions) {
		definitions = numDefinitions;
	}

	public int getDefinitions() {
		return definitions;
```

```
	}
}

public class Publisher {
	private String name;
	private Address address;

	public Publisher(String name, Address address) {
		this.name = name;
		this.address = address;
	}

	public String getName() {
		return name;
	}

	public Address getAddress() {
		return address;
	}
}

public class Address {
	private String streetAddress, city, state;
	private long zipCode;

	public Address(String street, String town, String st, long zip) {
		streetAddress = street;
		city = town;
		state = st;
		zipCode = zip;
	}

	public String toString() {
		String result;
		result = streetAddress + "\n";
		result += city + ", " + state + " " + zipCode;
		return result;
	}
}

public class Words {
	public Words() {}

	public static void main(String[] args) {
		Dictionary webster = new Dictionary();
		System.out.println("Number of pages: " +
		webster.getPages());
		System.out.println("Number of definitions: " +
		webster.getDefinitions());
		System.out.println("Definitions per page: " +
		webster.computeRatio());
	}
}
```

The full class diagram for the program above is shown below. Note that the `String` and other system classes are not included in the class diagram. Take time to identify and understand the dependency, aggregation, generalization, and realization relationships shown in the class diagram.



Sometimes, we may prefer to show a class diagram without fields and methods of classes and interfaces in a program. The following is such a class diagram for the same program. The key information that this diagram tries to show