**object**

* **An object:** is an *encapsulated* entity that contains **both**data and behaviors [that operate on that data].
  + In a properly designed object, data is local in scope
  + An object can restrict direct manipulation of its data by controlling access to both attributes (fields, local instances) and behaviors (methods) (i.e. *data hiding*)
* An object-oriented program can contain multiple copies of the same object, each with its own unique state configuration
* Relationships between objects can be compositional or inheritable

In object-oriented programming, the standard practice is to use accessor and mutator methods (commonly called "getters" and "setters") to interface with attributes.

**class**

A class is an abstract data type (ADT) that acts as the blueprint for an object. Classes define what the structure (attributes and methods) of an object will be; the object is then instantiated with specific attribute values (i.e. an object is a specific instance of a class). Classes exist at compile-time. Objects exist at run-time.

## **Encapsulation and Data Hiding**

The term **encapsulation** refers to the idea that an object has both data and behavior(methods) contained within the same abstraction.

The term data hiding refers to the idea that an object should be revealed only by the interfaces(refers to the methods) that other objects must have to interact with that object. Any details not required to use an object (including an object's implementation) should be hidden from all other objects.

An object's implementation consists of the attribute values and logic internal to each method, as well as any methods not included in the object's interface.

**Access Modifiers.**

Java defines four access modifiers: **private**, **protected**, **package**, and **public**.

* An attribute or method declared as **private** can only be accessed from **within the defining class.**
* An attribute or method declared as **protected** can be accessed from within the define **class** or any **subclasses**.
* An attribute or method declared without an **explicit modifier** can be accessed from within **any class in the package**.
* An attribute or method declared as **public** can be accessed **from anywhere within the program.**

## **Inheritance**

Inheritance is a method of abstraction that provides the ability to create a new class by using the attributes and behaviors of another class as a starting point and then specializing those methods (typically for a more specialized purpose).

* Inheritance is used when two objects have an "is-a" relationship between them.
  + A dog "is-a" animal
  + A plane "is-a" vehicle
  + A manager "is-a" employee
* Animal <|-- Dog inheritance (arrow )
* In these examples, "Animal" is referred to as the superclass and "Cat" and "Dog" are subclasses. Alternative terminology includes base/derived class, and parent/child class.

Anything marked as protected, default, or public in a superclass will be visible to all subclasses.

A good general guideline as to whether or not you should favor inheritance over composition is the Lisvok Substitution Principle (LSP). The LSP essentially says that any subclass you create should be able to be substituted for the superclass throughout the entire program. (i.e. if you have A <|-- B, then all instances of A can be replaced with B and not cause major issues.)

Lisvok substitution principle.

” if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.""

## **Polymorphism**

Polymorphism refers to the idea that code can be designed to work with values of multiple types. There are many different forms of polymorphism, but in this course we will primarily concern ourselves with parametric polymorphism and subtype polymorphism.

* In parametric polymorphism, code accepts the type as a parameter (either implicitly or explicitly).
  + - Java supports explicit parametric polymorphism through generics. The type tis passed as an object. ArrayList<Integer>. Integer is an object of the type int.

* In subtype polymorphism, code is designed to work with a Superclass, but the developer can override that behavior in a subclass, and the appropriate code will be executed at runtime.
  + - Java supports subtype polymorphism through inheritance.

When discussing object-oriented programming, the term polymorphism typically refers to subtype polymorphism, and parametric polymorphism is typically referred to as "generics".

## **Composition**

Composition refers to the idea that one class can contain a reference to another. Whereas inheritance abstracts the idea of an "is-a" relationship, composition abstracts the idea of a "has-a" relationship.

* In UML, composition can be modeled as either an aggregation relationship or a composition relationship.
* Aggregation implies that the component can exist without the container.
* Composition implies that the component cannot exist without the container.

From an implementation standpoint, this difference is very important. When two objects are related through composition, the **container object** is responsible for the creation and deletion of the component. When two objects are related through aggregation, the component may not necessarily subject to garbage collection just because the container has gone out of scope.

# **02 - How to Think in Terms of Objects**

To develop a good notion of the object-oriented thought process, you should:

* Know the difference between the interface and implementation
* Think more abstractly
* Provide the minimal interface possible

## **Knowing the Difference Between the Interface and the Implementation**

One of the keys to building a strong object-oriented design is to understand the difference between the interface and the implementation.

the interface is the set of services (e.g. methods) presented to the end user (i.e. another designer or developer)

The implementation is all of the algorithms, data structures, and other software components that support the interface.

The core idea behind separating the interface from the implementation is that, upon doing so, you can make changes to the implementation without changing the interface. This allows developers to change how the software works without impacting the end user.

What if you want the minimal interface possible for a class?

* You could start out by making everything private until an actual need arises. This need may arise either as you use the class within your own code, or as an end user attempts to use the class in theirs.

## **Using Abstract Thinking when Designing Interfaces**

Reuse is one of the main advantages of object-oriented programming. Interfaces that represent abstract (i.e. high-level or general) operations are generally easier to reuse than those that represent concrete operations.

## **Providing the Absolute Minimal User Interface**

* Give the users only what they absolutely need.
* Start off with a minimal interface and then add additional functionality through iterative development and agile practices.
* Do not make assumptions about what a user will need; only modify the interface when there is actual demand.
* Design classes from a user's perspective.
* Use user cases, user stories, and unit testing to help identify the minimal interface. Gather feedback from users of the system when possible.

Who are the users of these interfaces?

* Anyone that will be using the classes in their systems, including yourself.
* **It is the method signatures that define the interface, not their internal blocks.**

## **Constructors**

Constructors are special methods that are automatically called when a new object is instantiated from a class. In the Java language, this instantiation is done with the `new` keyword.

The role of a constructor is to set a new instance of the object into an initial state.

* This initial state does not need to be "null", in the sense that all instance variables must be zeroed out.

If a constructor does not have any parameters, then it is typically referred to as a **"default" constructor**.

The most important thing to note in this example is that the constructor has the same **identifier** (name) as the **class** and **does not have a return type**.

The second thing to notice is that the first thing is that our **constructor** call a method named ”super”.  The keyword ”super” acts as a reference (i.e. pointer) to an object's of the the superclass(super call the constructor of the supper class to initialize the instance variables of the super class . If there are no instance variables in the superclass, the super is left empty or is not needed). When used as a method, it allows us to explicitly call the object's superclass. When used as a method, it allows us to explicitly call the constructor of our superclass.

#### **The Default Constructor**

In most languages, including Java, you do not need to define your own constructor. If you do not explicitly define a constructor, then one is added automatically with the default behavior to first initialize any superclasses, and then to zero out all instance variables.

#### **Using Multiple Constructors**

What if you have a class with a large number of instance variables? One way to initialize this class would be provide a single constructor that takes in one parameter for each variable. However, an alternative approach is to leverage method overloading and define multiple constructors.

* Method overloading is the ability to define multiple methods with the same name, so long as their parameter lists differ.
* We have used the `this` keyword when assigning a value to our instance variables. Without it, we would need to give our parameters names that differ from the instance variables.
  + What is `this`? It's just a reference (i.e. pointer) to the object currently executing code.
* We have managed to duplicate a lot of code.

## **Error Handling**

there are four general courses of action:

* Ignore the issue
* Employ logic to check for the issue and abort the program if the issue occurs.
* Employ logic to check for the issue and try to fix the problem the issue occurs
* Assume the code works correctly and throw an exception if the issue occurs

#### **Exception Handling**

An exception is a class that represents an error or an unexpected event. Exceptions are instantiated from within problematic code and "thrown" our into the environment to be "caught" by the calling function and handled appropriately (by the caller). If an exception is not caught, then the software will crash.

Finally, exception handling should be reserved for exceptional circumstances, typically for situations where the results of the error may cause the program to crash or proceed with an invalid state.

As a closing remark, exception handling in Java is a little different than other languages.

Java provides two types of exceptions: checked and unchecked.

Checked exceptions can be check at the compile time. For example, failing to open a File can throw an exception. However, the unchecked exception cannot be checked at the compile time such as error in the code like arithmetic exception it mostly happen in the runtime.

## **The Importance of Scope**

* **Static methods** are methods that are shared between all instances of a class and can only access other **static methods** and **static variables**.
* **instance methods** are methods that are independent to each instance of a class.
* **class variables** are variables that are shared between all instances of a class; typically called "static" variables, they are accessible in both static and instance methods.
* **instance variables** are variables that are maintained independently **of each instance** of a class, but accessible by all instance methods.
* **local variables**are variables that are maintained independently within a block of code (or within a function, depending on the language); only accessible within the current block (or function)

## **Operator Overloading**

**Java does not support operator's overload.**

## **Multiple Inheritance**

Multiple inheritance refers to the idea that a class may have more than one superclass. Java gets around this problem by disallowing multiple inheritance. Instead, each class may have only one superclass, but may implement multiple **interfaces**.

# **04 - The Anatomy of a Class**

Consider the following source code for a Tree data structure (taken from the textbook).

To create meaningful identifiers for your own classes:

* Use intention-revealing names
  + You should not need a comment to explain what an identifier represents
* Avoid disinformation
  + Names should refer to functionality, not a specific implementation
* Make meaningful distinctions
  + Code is for the human to read. Don’t write code just to satisfy a compiler
  + Avoid terminology that doesn’t actually add value
    - E.g.: customer vs. customerData vs. customerInfo, NameString, pzWeight, etc
* Use pronounceable names
  + Code is the subject of conversation; you should be able to talk about it
* Use searchable names
* Avoid encodings (e.g. Hungarian notation, member prefixes)
  + Using encodings violates rules 2, 3, and 4
* Avoid mental mapping
  + Names should be related to the problem or solution domain
* Class names should have noun or noun phrases
* Method Names should have verb or verb phrases
* Don’t be cute
  + Clever names make it hard to understand what code does, and violate rules 1 and 7.
* Pick one word per concept
  + Avoid using synonyms for the same idea, e.g. “fetch”, “get”, “access”, “retrieve”. Just pick one and stick with it.
* Don’t pun
  + Do not use the same word for more than one purpose. It violates rules 2 and 3.
* Add meaningful context
  + Few words by themselves have a useful meaning. You must often combine words to create a “clean” identifier.
* Don’t add gratuitous context
  + In line with rule 6, don’t prefix variables by class names or redundant information.
  + Prefer short names to long names
    - Do you really need “getNumberOfSkinCareEligibleItemsWithinTransaction”? Maybe you should reconsider the class structure.
    - The only reasonable except is names for test cases when unit testing

# **05 - Class Design Guidelines**

## **Modeling Real-World Systems**

The public interface (refers to an object) to a class should be as small as possible while still conveying the full list of supported capabilities. These capabilities should be cohesive and support the idea of a single responsibility.

## **Designing Robust Constructors**

The purpose of a class' constructor is to place each instance into a valid initial state.

* If a default constructor is not appropriate for a class, then a parametrized constructor must be defined.
* Recall that public methods can be **overridden** within a **subclass**
* Private methods can be called within a **constructor**, but care needs to be taken to ensure they don't assume the object is fully defined at the time of their execution.

## **Designing Error Handling into a Class**

The following list of suggestions are generally useful when designing error handling mechanisms within a class:

* Error handling should not obscure the logic of your code
* Use exceptions instead of return codes (when possible)
  + Return codes force you to check for errors immediately after a function call
  + Exceptions let you isolate error handling, making it easy to see the real logic
* Write the **try/catch/finally** statements first
  + This helps define what errors should be expected by the code
* Avoid checked (compile time) exceptions
  + Many languages don’t have these now, but Java still does
  + **Checked exceptions violate the Open/Closed principle.** A change in low level detail can produce a change in the method signature.
* Provide context with **exceptions** to help isolate the cause of errors
* Define exception classes in terms of a caller’s needs
  + This may including **wrapping a third-party API to “clean up” the possible exceptions**
* Define objects and return values so that exceptions aren’t required to handle special cases
* Don’t return NULL – it forces rampant NULL checks
* Don’t pass NULL – it forces rampant NULL checks

The best approach to **error handling**, however, is to try and ensure that your deployed system has as few **errors as possible**. **Unit**, **integration**, and **regression testing** are three of the primary methods that facilitate this goal.

* **Unit Testing verifies the noticeable results of a *single* unit of work**
  + **Unit tests often use substitutes for the real dependencies in the system**
* **Integration Testing verifies the integration of multiple units of work**
  + Integration tests typically include real dependencies in the system
* **Regression Testing verifies that new modifications have not broken the previously tested system**
  + **Regression testing is typically done by running the entire test suite each time a change is made**

**A testing framework** is a **collection** of **libraries** that help developers automate their testing.

* **Testing frameworks automatically run your tests**
* Testing frameworks allow you to annotate tests
* Testing frameworks verify tests with Assert methods
  + Equality between two values
  + Thrown exceptions

Testing frameworks exists for most modern languages, with JUnit being the most popular testing framework for Java.

## **Designing with Reuse in Mind**

worthwhile to start off with a set of generalized behaviors and then create subclasses as required.

* This approach allows you to type your methods to a superclass and then pass in instances of a subclass. Polymorphism will ensure that the correct behavior is executed.

## **Designing with Maintainability in Mind**

Code changes over time, and someone will eventually need to make an update to your system.

## **Design for Object Persistence**

## **Unified Modeling Language (UML)**

## **Object Wrappers**

When these situations occur, one of the easiest ways to begin refactoring is to construct an **object wrapper**

Object wrappers are also frequently used to isolate non-portable code (i.e. code that depends on specific hardware or software configurations) and third-party libraries.

A compromise between these two approaches is to consult your design documents (e.g. product backlog and requirements) and to create an in initial model in UML before writing any code. This model can then be assessed and classes added or removed as necessary.

* You will frequently find that the process of modeling will help unlock additional insights that were overlooked during the initial analysis phase of your system.

## **Abstract Classes**

An **abstract** class is a class that contains abstract methods. An abstract method is a method that does not contain an **implementation at the time of declaration** and is denoted with a language-specific keyword (`abstract` in Java). An abstract class may also **contain non-abstract methods, and attributes**.

**Why would you want to avoid implementing a method?**

## **Interfaces**

an **interface** is a collection of method signatures, generally without a default **implementation**. If a class "uses" an interface, then it must provide an implementation for those methods. This creates a contract in the same was as an abstract class.

* In Java, an interface is distinct from a class, and defined with the `interface` keyword. This was done, in part, because Java does not support multiple inheritance.

## **The Dangers of Inheritance**

Inheritance is not without its faults: any changes made to the inheritable elements of a superclass will necessarily ripple through all subclasses.

* If a new public method is added to a superclass, all subclasses now have access to that method, regardless of whether or not the default implementation of that method is appropriate for a specific subclass. When this occurs, every subclass needs to be re-evaluated and modified accordingly.
* If a public class is removed from a superclass, then any polymorphic code that relied on that behavior will no longer work.
* If a variable in a superclass is changed from public to private, or private to public, it can have a similar impact to all subclasses.

To minimize the impact of future changes, inheritance should only be used when the relationship between two classes is a stick "is-a" relationship.

* If you find that you are overriding a superclass method so that it does "nothing", then that is a sign that your subclass is violating the "is-a" relationship.

# **08 - Mastering Composition**

## **Composition**

Composition models a "has-a" relationship between two classes and facilitates reuse storing object references within either the class or instance scope.

Composition allows you to decompose a class into an **aggregation** of simple parts. This often results in a system that is much easier to understand than one in which all data structures and logic are compressed into a single class.

Three types of composition: Association, Aggregation, and Composition. As with inheritance relationships, composition can also be represented in UML.

* **Association implies that the container uses the component in one form or another.**
* **Aggregation** implies that the component can exist without the container.
* **Composition** implies that the component cannot exist without the container.

![Diagram

Description automatically generated]()

From an implementation standpoint, these differences are very important. When two objects are related through **composition**, the **container** object is responsible for the creation and deletion of the **component**. When two objects are related through **association or aggregation**, the **component** may not necessarily subject to garbage collection just because the container **has gone out of scope**.

## **The Dangers of Composition**

Composition is not without its risks. As with inheritance, composition introduces dependencies - changes to a component may alter the behavior of the composing class. However, unlike inheritance, these dependencies are made explicit.

**Independency Injection**

To further increase the flexibility of our solution, we can use dependency injection to pass in the behaviors created in independent class through a **constructor** or a **mutator** method. For example:

**public** **class** **Bat** {

**private** EatingBehavior eatingBehavior; // class variables are objects defined in a separated class

**private** FlyingBehavior flyingBehavior;

**public** Bat() {// constructor were we need to initialize the objects with new.

**this**(**new** EatingBehavior(), **new** FlyingBehavior());

}

**public** Bat(EatingBehavior eatingBehavior) {

**this**(eatingBehavior, **new** FlyingBehavior());

}