Agile

Topics that you should be familiar with include:

* User Scenarios

describes a real-world example of how one or more people or organizations interact with a system.

**Scenario: ATM banking for the week.**

1. Sally Jones places her bank card into the ATM.
2. Sally successfully logs into the ATM using her personal identification number.
3. Sally deposits her weekly paycheck of $350 into her savings account.
4. Sally pays her phone bill of $75, her electric bill of $145, her cable bill of $55, and her water bill of $85 from her savings account
5. Sally attempts to withdraw $100 from her savings account for the weekend but discovers that she has insufficient funds
6. Sally withdraws $40 and gets her card back

* User Stories

A user story is an informal, general explanation of a software feature written from the perspective of the end user. Its purpose is to articulate how a software feature will provide value to the customer.

* Requirements

function or feature that a user needs. Requirements can be functions, constraints, business rules or other elements that must be present to meet the need of the intended users.

* Design
* Testing
* Scrum Events

The Sprint is a container for all other events. Each event in Scrum is a formal opportunity to inspect and adapt Scrum artifacts. These events are specifically designed to enable the transparency required. Failure to operate any events as prescribed results in lost opportunities to inspect and adapt. Events are used in Scrum to create regularity and to minimize the need for meetings not defined in Scrum. Optimally, all events are held at the same time and place to reduce complexity.

* Scrum Artifacts

Scrum’s artifacts represent work or value. They are designed to maximize transparency of key information. Thus, everyone inspecting them has the same basis for adaptation.

Each artifact contains a commitment to ensure it provides information that enhances transparency and focus against which progress can be measured:

* For the Product Backlog it is the Product Goal.
* For the Sprint Backlog it is the Sprint Goal.
* For the Increment it is the Definition of Done.

These commitments exist to reinforce empiricism and the Scrum values for the Scrum Team and their stakeholders.

### **Product Backlog**

The Product Backlog is an emergent, ordered list of what is needed to improve the product. It is the single source of work undertaken by the Scrum Team.

Product Backlog items that can be Done by the Scrum Team within one Sprint are deemed ready for selection in a Sprint Planning event. They usually acquire this degree of transparency after refining activities. Product Backlog refinement is the act of breaking down and further defining Product Backlog items into smaller more precise items. This is an ongoing activity to add details, such as a description, order, and size. Attributes often vary with the domain of work.

The Developers who will be doing the work are responsible for the sizing. The Product Owner may influence the Developers by helping them understand and select trade-offs.

Kanban Board

A kanban board is an agile project management tool designed to help visualize work, limit work-in-progress, and maximize efficiency (or flow). It can help both agile and DevOps teams establish order in their daily work. Kanban boards use cards, columns, and continuous improvement to help technology and service teams commit to the right amount of work, and get it done!

# 01 - Introduction to Object-Oriented Concepts

## **The Fundamental Object-Oriented Concepts**

The fundamental concepts that we're going to talk about (at a high level) today, are:

* Encapsulation
* Inheritance
* Polymorphism
* Composition

But first, some preliminaries.

## **Object-Oriented Programming and Legacy Systems**

The world is filled with non-OO systems that work perfectly fine. These systems are often referred to as Legacy Systems

* But, not all legacy systems are non-OO

Object-oriented code is not meant to outright replace existing legacy systems, but object-oriented code can be very useful when creating a new system or integrating legacy code into an object-oriented system.

* Object wrappers are a common technique to integrate legacy code into object-oriented systems

## **Procedural Versus Object-Oriented Programming**

In procedural programing, computation occurs by passing globally defined data structures into carefully constructed functions and procedures that act as black boxes which transform those incoming data structures.

* Data is global in scope and can be accessed by any function designed to understand that data
  + This can lead to data corruption or manipulation by adversarial programs
  + Testing and debugging can be complicated because you cannot be entirely sure where the data might be modified
* Relationships between data structures and functions are compositional
* Sometimes developers place related data and the functions that work on that data into a singular module
  + This approach offers many benefits, but a module is not exactly an "object", and this approach is not "object-oriented"

In an object-oriented program, entities called "objects" are the primary method of computation, and their interactions are the basis for complex behaviors.

* 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 and behaviors (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

The key difference between these approaches is in how one thinks about program design.

It is, of course, possible to design bad objects that act almost entirely like procedural programs, but this is not a suggested practice. Furthermore, you can [write object-oriented code in a procedural language, like C (Links to an external site.)](https://www.cs.rit.edu/~ats/books/ooc.pdf).

## **What Exactly Is an Object?**

Objects are abstractions that act as the building blocks of object-oriented programs. Objects are instantiated from a class and contain both data and behaviors.

The data stored within an object represents the state of that object. This state is defined by a collection of one or more attributes.

* If you consider a "Person" object, attributes might be name, age, height, weight, eye color, etc. A simulation game may contain multiple "Person" objects, each with different attribute values.

The behaviors stored within an object represent what that object can do. In object-oriented terminology, these behaviors are called methods, and they are invoked on an object via message passing.

* Object A passes a message to Object B to make a request for Object B to invoke it's "foo" method

One of the key differences between object-oriented programing and procedural programming is the level of data hiding.

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

* By wrapping attribute access, it is possible to change an object's implementation without changing its interface

[UML daigrams (Links to an external site.)](https://www.linkedin.com/learning/software-design-modeling-with-uml/) (especially Class Diagrams and Object Diagrams) are a common way to visualize objects for discussion.

## **What Exactly is a Class?**

A class is an abstract data type 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.

This may easier to understand by looking at some code.

**public** **class** **Person** {

**private** String name;

**private** String address;

name and address are attbibutes.

**public** String getName() {

**return** name;

}

**public** void setName(String value) {

name = value;

}

**public** String getAddress() {

**return** address;

}

**public** void setAddress(String value) {

address = value;

}

}

In this example, we see that the class Person is defined to have two attributes (name and address) and four methods (getName, setName, getAddress, and setAddress).

Once we have defined a class, we can then use that class to create objects. Computation is the result of those objects sending messages to one another.

**public** **class** **App** {

**public** **static** void main(String[] args) {

Person john = **new** Person();

john.setName("John");

john.setAddress("123 Cookie Drive");

Person erica = **new** Person();

erica.setName("Erica");

erica.setAddress("42 Cupcake Lane");

System.out.println(john.getName() + " knows " + erica.getName());

}

}

(Don't worry too much about the specific Java syntax right now. It will make more sense as we get into the course.)

### **Using Class Diagrams**

In this course, we will use [PlantUML (Links to an external site.)](https://plantuml.com/" \t "_blank) as our diagramming tool. Using PlantUML, we can create a class diagram of our Person class:

@startuml  
class Person {  
 name: String  
 address: String  
  
 getName(): String  
 setName(String)  
 getAddress(): String  
 setAddress(String)  
}  
@endeml

## **Encapsulation and Data Hiding**

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

The term data hiding refers to the idea that an object should reveal only the interfaces 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 interface defines the list of accepted messages for an object. From a practical standpoint, an object's interface is the list of pubic method signatures (where a method signature consists of the return value, function name, and parameter types).

* Direct access to an object's attributes should almost never be provided by the interface. Reading and writing attribute values should be delegated to accessor and mutator methods to enable validation and reduce coupling.

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.

Why is it important to support data hiding by separating the interface from the implementation? So that future changes to an object's algorithms can be invisible to the user. Given a well thought-out interface, the author of an object can update their code without impacting the users of that object.

* In the real world, we might consider an electrical outlet in your apartment as part of the interface to a powerplant. To use the powerplant, you just plug in a device - there's no need to know whether that power is generated by wind, solar, coal, nuclear, etc.

So how does data hiding work in [Java (Links to an external site.)](https://docs.oracle.com/javase/specs/jls/se16/jls16.pdf)? In Java, you denote each attribute and method with an access modifier. 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.**

(If some of these terms are unfamiliar, do not worry - we will cover them as we progress.)

## **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

Using PlantUML, we can express a very basic inheritance relationship with:

@startuml

class Animal

class Dog

class Cat

Animal <|-- Dog

Animal <|-- Cat

@enduml

This diagram can then be translated into the following Java code:

**public** **class** **Animal** { }

**public** **class** **Dog** **extends** Animal { }

**public** **class** **Cat** **extends** Animal { }

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.

The previous example was exceptionally trivial. The real power of inheritance comes when we begin to include non-private attributes and behaviors in the superclass.

* Non-private? OOP languages are somewhat inconsistent in how they denote which attributes and behaviors are visible in a subclass. Recall that in the case of Java, there are four access modifiers: private, protected, default, and public. Of these four, anything marked as protected, default, or public in a superclass will be visible to all subclasses.

Consider the following example:

@startuml

class Animal {

- sound: String

+ makeSound(): String

}

class Dog {

+ Dog()

}

class Cat {

+ Cat()

}

Animal <|-- Dog

Animal <|-- Cat

@enduml

with the corresponding Java program:

*// Animal.java*

**public** **class** **Animal** {

String sound = "";

**public** String makeSound() {

**return** sound;

}

}

*// Cat.java*

**public** **class** **Cat** **extends** Animal {

**public** Cat() {

sound = "meow";

}

}

*// Dog.java*

**public** **class** **Dog** **extends** Animal {

**public** Dog() {

sound = "woof";

}

}

*// App.java*

**public** **class** **App** {

**public** **static** void main(String[] args) {

Cat aCat = **new** Cat();

Dog aDog = **new** Dog();

System.out.println(aCat.makeSound());

System.out.println(aDog.makeSound());

}

}

 Notice that in the implementation, neither `Cat` nor `Dog` define the variable `sound` or the method `makeSound`. They inherit these from their superclass, `Animal`. These inherited elements are then used from within a simple driver program.

* In this course, we will generally prefer automated unit tests written in JUnit over driver programs that test code via the `main` method.

### **A note on inheritance**

Over the years, many developers (both professional and novice) have abused inheritance as a way to enforce the DRY principle (don't repeat yourself). This practice leads to overly complicated inheritance hierarchies that impose artificial relationships, subclasses with overridden methods that behavior fundamentally different from their superclass, and subclasses with numerous behaviors that are not defined in their parent classes. The end result of this practice is software that is fragile and difficult to maintain.

Due to these abuses, modern software engineering has come to favor composition over inheritance. Inheritance is still used, but it is typically reserved for those cases where the relationship between two classes truly is an "is-a" relationship. 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.)

## **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.
* 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".

Consider the following class structure:

@startuml

class Animal {

+ makeSound(): String

}

class Dog {  
 + makeSound(): String   
}

class Cat {  
 + makeSound(): String   
}

Animal <|-- Dog

Animal <|-- Cat

@enduml

with the corresponding Java program:

*// Animal.java*

**public** **class** **Animal** {

**public** String makeSound() {

**return** "";

}

}

*// Cat.java*

**public** **class** **Cat** **extends** Animal {

**public** String makeSound() {

**return** "meow";

}

}

*// Dog.java*

**public** **class** **Dog** **extends** Animal {

**public** String makeSound() {

**return** "woof";

}

}

*// App.java*

**public** **class** **App** {

**public** **static** void main(String[] args) {

Animal aCat = **new** Cat();

Animal aDog = **new** Dog();

System.out.println(aCat.makeSound());

System.out.println(aDog.makeSound());

}

}

This is an example of subtype polymorphism. Notice that our variables are declared to be of type `Animal` in the `App` class, yet they are initialized with the subclasses of `Cat` and `Dog`. Also notice that `makeSound` is defined with the same signature in all classes of our `Animal` hierarchy. When our `main` method is run, the appropriate version of `makeSound` is executed and each call produces a different return value, despite both variables being declared to be the same type.

## **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.

Consider the following class structure that we might use to start modeling a Gradebook application:

@startuml

class Gradebook {

instructor: Professor

students: ArrayList

}

class Professor {

name: String

}

class Student {

assignments: ArrayList

}

class Assignment {

directions: String

submission: String

grade: char

}

Gradebook "1" \*--> "1" Professor : contains

Gradebook "1" \*--> "1..\*" Student : contains

Student "1" \*--> "0..\*" Assignment : contains

@enduml

In this example, we see that the gradebook is "composed" of one `Professor` object and multiple `Student` objects. Similarly, a `Student` is composed of multiple `Assignment` objects.

* 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.

## **What You Should Know**

Concepts:

* How does procedural programming differ from object-oriented programming?
* What is an object?
* What is a class?
* What is a class diagram?
* What is encapsulation?
* What is inheritance?
* What is polymorphism?
* What is composition?

Syntax

* How to work with structured code (assignment, arithmetic and relational operators, if/else statements, loops, functions)
* How to define a simple class with public methods and private attributes.
* How to assign a new class instance to a variable

# 02 - How to Think in Terms of Objects

Migrating from a procedural language, such as C, to an object-oriented language, such as Java, requires not just the learning of new syntax; but, the learning of an entire new paradigm of problem solving. To write good object-oriented software, you must develop an object-oriented thought process.

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.

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

* The interface in the context of software construction should not be confused with a Graphical User Interface (GUI) or text-based interface.
* What services should be included in the interface? Only what is strictly necessary to use the component. This should be determined by a use case analysis and the software design documentation.

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.

Some real life cases of interface/implementation separation include:

* Making a phone call: you dial a number and the phone rings without you needing to worry about if the phone uses a landline, cellular, or VOIP connection.
* Driving a car: you turn the key, move the gear shifter, press the peddles, and steer the car without needing to worry about how the engine works
* Making toast: you insert the bread, press the button, then pull out the toast; all without worrying how the heating mechanism works or where the power is coming from

As an example in the context of an object-oriented program, consider an interface to a simple database reader class.

@startuml

class DatabaseReader {

+ openConnection(string): void

+ closeConnection(): void

+ gotoFirstRecord(): void

+ gotoLastRecord(): void

+ getRecordCount(): int

+ hasMoreRecords(): boolean

+ gotoRecord(string): void

+ getCurrentRecord(): Record

+ getNextRecord(): Record

}

@enduml

In this example, we see that only the public methods are listed. There may be private or protected methods, or even attributes, but they would be part of the implementation, not the interface.

* To use this class, do we need to know what sort of database is being used or not connections are handled? No.
* To use this class, do we need to know how the database handles cursor positioning? No.
* To use this class, do we need to know how the database keeps track of how many records there are? No.
* All we care about is the ability to read data from the database.

If we wanted to use this DatabaseReader class within our own application, we need only create an instance and then call the appropriate methods.

**public** **class** **App** {

**public** **static** void main(String[] args) {

DatabaseReader reader = **new** DatabaseReader();

reader.openConnection("sql://path/to/my.db");

*// use the reader as needed*

reader.closeConnection();

}

}

Is it really worthwhile to create your own class to access a database? Why not just use the database API directly?

* Creating a wrapper around a third-party API is always a good idea. Wrapper classes provide an interface around foreign code, allowing that code to be swapped out without requiring users of your system to change their own code.

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.

* "Take my to the airport" vs. "Pull out here, make a left, stop at the next light, right right onto the highway for 10 miles"

Regardless of the generality of the interface, the interface itself must be cohesive (i.e. all of the methods provided are directly relevant to the class).

* If you find that your interface lacks cohesion and seems to be providing two or more distinct groups of operations, then that is a clear sign that you probably need to break your single class up into multiple classes.

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

The general rule when designing a class is to provide the user with as little knowledge of the implementation as possible. There are few rules that can help accomplish this goal:

* 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.

Anything not declared public is automatically part of the implementation. However, real code generally includes implementation details inside of public functions.

As an example, consider the following class in which we explicitly separate the interface from the implementation:

**public** **class** **Response** {

**public** double calculateResponse(int code) {

**return** \_calculateResponse(code);

}

**private** double \_calculateResponse(int code) {

**return** code \* 2 - Math.sqrt(code);

}

}

This approach probably feels a bit "off" to an experienced software developer.

Next, consider an alternative design in which we incorporate the implementation directly into the method being provided as part of the interface:

**public** **class** **Response** {

**public** double calculateResponse(int code) {

**return** code \* 2 - Math.sqrt(code);

}

}

This approach resembles how most people write their code.

In both examples, the implementation is cleanly separated from the interface as we can change the internal code without needing to alter the method signature.

**It is the method signatures that define the interface, not their internal blocks.**

## **What You Should Know**

Concepts

* How does an object's the interface differ from it's implementation?
* How can abstract thinking aid in designing an object's interface?
* How can you design a minimal interface for an object?

Syntax

* How to use the `public`, `private`, `protected`, and package access modifiers to separate a class' interface and implementation

# 03 - More Object-Oriented Concepts

We previously discussed the ideas of abstraction, encapsulation, polymorphism, inheritance, and composition. We now bring our attention to some of the concepts critical to the implementation of object-oriented software.

## **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 follow example displays what a "default" constructor might look like in the Java language:

**public** **class** **Employee** {

**private** String firstName;

**private** String lastName;

**private** double salary;

**public** Employee() {

super();

firstName = "unknown";

lastName = "entity";

salary = -1.0;

}

}

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

The second thing to notice is that the first thing our constructor does is to call a method named `super`.  The keyword `super` acts as a reference (i.e. pointer) to an 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.

It is generally a good idea to always define your own constructors. Defining your own default constructor improves the maintainability of your code by explicitly stating what should happen, even if those actions are trivial.

#### **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.

The following example shows how this can be done in Java:

**public** **class** **Employee** {

**private** String firstName;

**private** String lastName;

**private** double salary;

**public** Employee() {

**super**();

firstName = "";

lastName = "";

salary = 0;

}

**public** Employee(String firstName, String lastName) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.salary = 0;

}

**public** Employee(String firstName, String lastName, double salary) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.salary = salary;

}

}

There are two things of note in this example.

1. 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.
2. We have managed to duplicate a lot of code.

In general, we would like to minimize code duplication. Duplicate code is hard to maintain and can easily be the source of bugs (especially if it gets there via copy/paste). To reduce the code duplication in Java, we can make use of `this` to call a different constructor within our object.

**public** **class** **Employee** {

**private** String firstName;

**private** String lastName;

**private** double salary;

**public** Employee() {

**this**("", "", 0);

}

**public** Employee(String firstName, String lastName) {

**this**(firstName, lastName, 0);;

}

**public** Employee(String firstName, String lastName, double salary) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

**this**.salary = salary;

}

**public** String getFirstName() {

**return** **this**.firstName;

}

}

## **Error Handling**

Any software of non-trivial size is almost guaranteed to have software "bugs". When these bugs occur, 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 if it occurs
* Assume the code works correctly and throw an exception if the issue occurs

As a general rule of thumb, never ignore the issue. Allowing software to continue to run in an invalid state will at best cause a minor annoyance, and at worse, result in injury to people or property.

The second and third approach use standard "if/else" logic to detect issues and deal with them accordingly.

Crashing the software when a issue occurs is a safe option, but it will inevitably frustrate users. This is not necessarily a bad technique to use when testing your software, although any crashes should be accompanied by logging information.

Attempting to recover when a issue occurs can be an acceptable solution if the user is allowed to correct the program state. Attempting to assign "default" values without user interaction may produce a working program, but the output may be unanticipated and lead to user frustration. If the user is unable to provide input with regards to the correct state, then it is typically better to just crash the program.

The fourth option on the list is referred to as exception handling.

#### **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.

Consider an example in which we attempt to divide by 0:

*// ExceptionExample.java*

**public** **class** **ExceptionExample** {

**public** **static** double DivisionExample(int x, int y) {

double ans = 0;

**try** {

ans = x / y;

} **catch**(ArithmeticException e) {

*// rethrow the exception instead of handling it*

**throw** **new** IllegalArgumentException("y = 0");

}

**return** ans;

}

}

*// ExceptionExampleTest.java*

**class** **ExceptionExampleTest** {

@Test

**public** void division\_example\_should\_return\_answer\_with\_valid\_input() {

double actual = ExceptionExample.DivisionExample(10, 1);

double expected = 10;

assertEquals(expected, actual);

}

@Test

**public** void division\_example\_should\_except\_with\_zero\_divisor() {

assertThrows(IllegalArgumentException.class, () -> ExceptionExample.DivisionExample(10, 0));

}

}

Here, we see that attempt to do the division regardless of the value of `y`, If there is a problem, such as `y = 0`, then we will throw an exception that the caller will need to handle.  We don't especially care about how the caller handles it, that's not our responsibility. To ensure that our function works as we expect, we also construct a pair of simple test cases to validate the behavior in both the positive case and the negative case.

As a point of comparison, let us consider an alternative approach to this behavior in which we use an if/else statement.

*// ExceptionExample.java*

**public** **class** **ExceptionExample** {

**public** **static** double DivisionExample(int x, int y) {

**if**(y != 0) {

**return** x / y;

} **else** {

**return** Double.MAX\_VALUE;

}

}

}

*// ExceptionExampleTest.java*

**class** **ExceptionExampleTest** {

@Test

**public** void division\_example\_should\_return\_answer\_with\_valid\_input() {

double actual = ExceptionExample.DivisionExample(10, 1);

double expected = 10;

assertEquals(expected, actual);

}

@Test

**public** void division\_example\_should\_return\_error\_value\_with\_valid\_input() {

double actual = ExceptionExample.DivisionExample(10, 0);

double expected = Double.MAX\_VALUE;

assertEquals(expected, actual, 0.01);

}

}

The primary thing to note in this alternate example is that we are returning a sentinel value if `y=0`. This is dangerous, as the value chosen could very well be the correct answer when `y != 0`. Exceptions allow us to avoid this situation entirely. You may also notice that the code is easier to read and reason about when using exceptions, as there is no need to trace out the conditional logic to determine which branch represents success and which branch represents failure.

* In a language that allows access to pointers, such as C, you can use an "output parameter" to store a boolean value that represents success.

It should also be noted that, while exceptions provide a more "object-oriented" way to conduct error handling, they are not required for a language to be considered "object-oriented" - every try/catch block can be replaced by if/else logic at the cost of readability.

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. If/else logic should still be used where it makes the most sense (e.g, looping to read and validate user input), but as soon as an error code is introduced, it's time to consider using an exception.

As a closing remark, exception handling in Java is a little different than other languages. Java provides two types of exceptions: checked and unchecked. Make sure you read the material on exception handling and familiarize yourself with the details

## **The Importance of Scope**

The visibility of a method or variable within a program is determined by its 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)

Be advised that some languages (e.g. ruby and python) also have "class methods" that are sort of like static methods, but they have implicit access to the class in the same way that instance methods have implicit access to the instance.

## **Operator Overloading**

In some languages, operators such as +, -, \*, /, (), >, <, etc. are mapped to functions that can be overloaded by the user. This allows one to write code that is easier to read in some ways, but potentially confusing in others. For example, given two vectors `x` and `y`, it makes more sense to say `x + y` than it does to say `x.add(y)`; however, the code becomes very confusing if someone redefines `+` to work in a way that is semantically inconsistent with the notion of "addition".

## **Multiple Inheritance**

Multiple inheritance refers to the idea that a class may have more than one superclass. However, there is a danger in this approach: when multiple superclasses define a method with the same signature, the subclass will not know which version to use. This scenario is lovingly referred to as "the deadly diamond of death".

Java gets around this problem by disallowing multiple inheritance. Instead, each class may have only one superclass, but may implement multiple interfaces (not to be confused with "interface" as in the collection of publicly available method signatures). Refer to the Java language specification and documentation for additional details.

* Essentially, java differentiates between an abstract class and an interface.

## **Object Operations**

Most object-oriented languages provide an "object" superclass that all other objects inherit from, either directly or indirectly. The specific methods defined in this `Object` class vary from language to language, but typical operations include equality, hashing, string conversion, and copying.

* Copying an object and comparing equality between two objects must account for references to any component objects. As a general rule of thumb, you almost always want to define your own equality and copying methods to ensure that you obtain the correct results.

## **What You Should Know**

Concepts

* What default and parameterized constructors?
* How can you use if/else and try/catch/finally statements to handle errors?
* What are the different levels of scope?
* What is operator overloading?
* What is multiple inheritance, and why can it be problematic?
* What are some common operations that all objects have in common?

Syntax

* How to define a default and parameterized constructors
* How to inherit from a single class
* How to use the `this` and `super` keywords
* The difference between checked and unchecked exceptions
* How to use try/catch/finally blocks to handle exceptions
* How do define custom exceptions
* How to define static methods and static variables

# 04 - The Anatomy of a Class

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

*// TreeNode and Tree class declarations for a binary search tree.*

**package** **com.deitel.datastructures**;

*// class TreeNode definition*

**class** **TreeNode**<E **extends** Comparable<E>> {

*// package access members*

TreeNode<E> leftNode;

E data; *// node value*

TreeNode<E> rightNode;

*// constructor initializes data and makes this a leaf node*

**public** TreeNode(E nodeData) {

data = nodeData;

leftNode = rightNode = **null**; *// node has no children*

}

*// locate insertion point and insert new node; ignore duplicate values*

**public** void insert(E insertValue) {

*// insert in left subtree*

**if** (insertValue.compareTo(data) < 0) {

*// insert new TreeNode*

**if** (leftNode == **null**) {

leftNode = **new** TreeNode<E>(insertValue);

}

**else** { *// continue traversing left subtree recursively*

leftNode.insert(insertValue);

}

}

*// insert in right subtree*

**else** **if** (insertValue.compareTo(data) > 0) {

*// insert new TreeNode*

**if** (rightNode == **null**) {

rightNode = **new** TreeNode<E>(insertValue);

}

**else** { *// continue traversing right subtree recursively*

rightNode.insert(insertValue);

}

}

}

}

*// class Tree definition*

**public** **class** **Tree**<E **extends** Comparable<E>> {

**private** TreeNode<E> root;

*// constructor initializes an empty Tree of integers*

**public** Tree() {root = **null**;}

*// insert a new node in the binary search tree*

**public** void insertNode(E insertValue) {

**if** (root == **null**) {

root = **new** TreeNode<E>(insertValue); *// create root node*

}

**else** {

root.insert(insertValue); *// call the insert method*

}

}

*// begin preorder traversal*

**public** void preorderTraversal() {preorderHelper(root);}

*// recursive method to perform preorder traversal*

**private** void preorderHelper(TreeNode<E> node) {

**if** (node == **null**) {

**return**;

}

System.out.printf("%s ", node.data); *// output node data*

preorderHelper(node.leftNode); *// traverse left subtree*

preorderHelper(node.rightNode); *// traverse right subtree*

}

*// begin inorder traversal*

**public** void inorderTraversal() {inorderHelper(root);}

*// recursive method to perform inorder traversal*

**private** void inorderHelper(TreeNode<E> node) {

**if** (node == **null**) {

**return**;

}

inorderHelper(node.leftNode); *// traverse left subtree*

System.out.printf("%s ", node.data); *// output node data*

inorderHelper(node.rightNode); *// traverse right subtree*

}

*// begin postorder traversal*

**public** void postorderTraversal() {postorderHelper(root);}

*// recursive method to perform postorder traversal*

**private** void postorderHelper(TreeNode<E> node) {

**if** (node == **null**) {

**return**;

}

postorderHelper(node.leftNode); *// traverse left subtree*

postorderHelper(node.rightNode); *// traverse right subtree*

System.out.printf("%s ", node.data); *// output node data*

}

}

The identifiers for the two classes being defined are `TreeNode` and `Tree`. The importance of a good class identifier (i.e. name) cannot be understated. A good identifier is descriptive. It provides immediate information about what a class does, and potentially how it interacts within the larger system.

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

There are also a number of comments in the code, not all of which are necessary. Comments can be useful when writing examples for novice programmers, but they do not make up for bad code. Explain yourself in code by using meaningful identifiers and functional decomposition.

Good Comments:

* Legal comments
* Informative comments (if its impossible to make the code readable)
* Explanation of intent
* Clarification (mainly when working with code you don’t control)
* Warning of consequences
* TODO comments
* Describing an API for public consumption (e.g. Javadoc)
* Documenting pre- and post-conditions of a function

Bad Comments

* Comments for the sake of comments (e.g. “required” by management)
* Position markers (banners, separators, etc.)
* Close brace comments
* Wrong or misleading comments
* Comments that duplicate code
* These run rampant through javadocs, where the function signature is sufficient
* Comments that track file modification (just use version control)
* Rambling comments that state opinions
* Information better maintained in source code control
* Commented out code
* Just remove it; leaving it leads to confusion and version control systems let you undo

The attributes in `TreeNode` are given package visibility, whereas the attribute of `Tree` is private. At first glance, the use of package access might be questionable, but if we carefully inspect the `Tree` class we see that there are no accessor methods for the `root` variable, and thus no apparent way to gain access to the `TreeNode` instances. There is, however, a mutator method in the form of `insertNode`. Pay special attention to how the user provides a value to be inserted into the tree, and not an actual `TreeNode` object.

The constructors for these classes are fairly standard. The `TreeNode` constructor allows the user to pass in a value and stores it directly. The `Tree` constructor is a default constructor that ensures the `root` of the tree is `null`. The `Tree` is only useful after the user calls the `insertNode` method.

Both of these classes provide a public interface, as defined by the methods declared as `public`. The `tree` class also makes use of private implementation methods, defined by those methods declared as `private`. Private implementation methods are an important tool for improving the readability and maintainability of your code.

To improve the maintainability of your code, there are a number of guidelines that should be followed when defining methods (either public interfaces or private implementations):

* Small – as small as possible
  + Most code within a block can (and should) be put into its own function
  + Small functions help document code
* Do one thing
  + When functions do one thing, they are small by nature
* One level of abstraction per function
  + Code should be able to read top-down, with each function call referring to a slightly more concrete idea.
* Favor polymorphism over “switch” statements
* Use descriptive names
* Minimize the number of function arguments: 0 > 1 > 2 >>> 3+
  + Create “method objects” and “argument objects” as an alternative
  + Use default and keyword arguments when possible
  + Instead of using “flags” as parameters, make one function for each flag value
* Prefer returning values to modifying arguments
* Avoid side effects – they are lies
* Enforce "Command Query Separation"
  + Functions should either do something or answer something, but not both
* Favor exceptions over returning error codes
  + Error codes must be handled immediately, but exceptions allow you separate the behavior from the error handling (and remember that error handling is one “thing”)

## **What You Should Know**

Concepts

* What makes a good identifier?
* What makes a bad identifier?
* What makes a good comment?
* What makes a bad comment?
* When and why would you use one access modifier over another?
* How does a class attribute differ from an instance attribute?
* Why do you define a constructor?
* What is an accessor method?
* Why would you define an accessor method?
* What is a mutator method?
* Why would you define a mutator method?
* What defines the public interface of a class?
* Why do you want to separate a class' public interface from its private implementation?
* How does a class method differ from an instance method?
* How can you improve the maintainability of your methods?
* What are local variables and when should you use them?
* What are the different ways in which you can handle runtime errors within your code?

Syntax

* By this point in the course, you should be able to write simple Java programs using user-defined objects. You should know how to use inheritance and composition, but not necessarily why. If you are still struggling with these basic concepts, you will need to review the supplemental material provided in the earlier course modules.

# 05 - Class Design Guidelines

## **Modeling Real-World Systems**

The initial approach to problem solving in object-oriented programming often takes the form of a bottom-up analysis, wherein the primary objects and their interactions with each other are identified and defined. This approach is, in contrast to a top-down approach, taken with procedural languages, in which data and behavior are logically separate entities and the solution is viewed as a sequence of events. Top-down analysis still has its place in OOP, but it occurs primarily within the objects themselves.

This bottom-up approach can make it easier to model real-world objects, since the developers are able to think at a higher level of abstraction and more easily incorporate domain knowledge.

## **Identifying the Public Interfaces**

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.

* If the interface is missing necessary methods, then the class will feel incomplete and require the user to attempt to fill in the gaps.
  + Inheritance and composition provide reasonable methods to extend a class, but if every user extends your class in the same way, it's a sign that your design is flawed.
* If the interface is has unnecessary methods, then the class will feel inconsistent and risk making the user's code harder to maintain.

User scenarios, user stories, and requirements provide the starting point for a class' public interface.

It is important to keep in mind that not every user a commercial customer. Some of the most common users of your software components are other developers in your organization (including quality assurance engineers), or in the case of open source software, all of the external contributors.

## **Designing Robust Constructors**

The purpose of a class' constructor is to place each instance into a valid initial state. You should almost always define your own constructor when creating an object, instead of relying on the underlying language's default constructor.

* If a default constructor is not appropriate for a class, then a parametrized constructor must be defined.

You should generally avoid calling public methods of a class from within a constructor; there is a chance that a subclass may have redefined the method and calling that new version before the object is initialized may cause issues.

* 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.

If a class contains instance variables to other objects [by way of composition], then it may be more appropriate to require the user to pass in an instance of those objects rather than having the class itself instantiate those components.

* If a class instantiates a component, it is responsible for memory manage of that object; it also limits the ability to leverage polymorphism since the constructor - not the user - is responsible for creating the concrete class.
* If a class allows a reference to be passed in via the constructor, then the caller is responsible for that object's memory management; but, the caller can also pass in any subtype of the expected parameter that conforms to the same interface.

If a class contains instance variables to other object types, be wary of circular references (e.g. A-B-A). Many modern languages, such as Java, employ garbage collectors for automatic memory management; however, circular references (especially indirect circular references: A-B-C-A) can still result in memory leaks.

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

As a general rule, your application should only crash as a last resort. Proper error handling can increase a class' robustness by allowing the system to recover from unforeseen circumstances.

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**

What makes a good test?

* **Automated and repeatable**
* Easy to implement
* Relevant in the future
* Easy to run
* Run quickly
* Consistency in the results
* Full control of the unit under test
* Fully isolated from other tests
* Highlights the cause of failure

**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**

The higher the abstraction level of a class, the more likely it can be reused. Eventually you will need to get specific, but it may be 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 Extensibility in Mind**

When designing a class, keep the single responsibility principle (SRP) at the forefront of your thinking. If your class abides by the SRP, then future developers can use inheritance or composition to easily extend the behavior of your classes.

* If your classes do not follow the SRP and have a low degree of cohesion, then future developers will not have a clear sense of what sort of behaviors they can (or should) be adding.
* If your classes are tightly coupled, then any future extensions may run the risk of introducing a bug by altering the behavior of an unknown method through polymorphism.

To further enhance the extensibility of a class, it is often useful to define methods to copy, compare, and generate a string representation. These methods can then be overridden in a subclass to account for new data without needing to know the specifics of the superclass.

## **Designing with Maintainability in Mind**

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

Use a coding standard to ensure consistent formatting across your system. Many languages already have multiple coding stands to choose from, for example:

* NASA C Style - [http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf (Links to an external site.)](http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf)
* Linux Kernel C Style - [https://www.kernel.org/doc/Documentation/CodingStyle (Links to an external site.)](https://www.kernel.org/doc/Documentation/CodingStyle)
* GNU C Style - [https://www.gnu.org/prep/standards/html\_node/Writing-C.html (Links to an external site.)](https://www.gnu.org/prep/standards/html_node/Writing-C.html)
* Google C++ Style - [https://google.github.io/styleguide/cppguide.html (Links to an external site.)](https://google.github.io/styleguide/cppguide.html)
* OpenOffice C++ Style - [https://wiki.openoffice.org/wiki/Cpp\_Coding\_Standards (Links to an external site.)](https://wiki.openoffice.org/wiki/Cpp_Coding_Standards)
* PEP 8 – Style Guide for Python - [https://www.python.org/dev/peps/pep-0008/ (Links to an external site.)](https://www.python.org/dev/peps/pep-0008/)
* Google Python Style - [https://google.github.io/styleguide/pyguide.html (Links to an external site.)](https://google.github.io/styleguide/pyguide.html)
* Google Java Style - [https://google.github.io/styleguide/javaguide.html (Links to an external site.)](https://google.github.io/styleguide/javaguide.html)
* Apache ACE Java Style - [https://ace.apache.org/docs/coding-standards.html (Links to an external site.)](https://ace.apache.org/docs/coding-standards.html)

Ensure that all identifiers in your code are descriptive and reflect their purpose.

* 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

Classes must be designed with maintainability in mind to ensure that these future changes can be done smoothly.

* Classes should be small and follow the Single Responsibility Principle
* Classes should be be cohesive
  + Keep the number of instance variables small
  + Every method should use one or more of a classes instance variables
  + If a subset of variables are only used by a few methods, it’s a sign that you might be able to extract those methods into a separate class
* Classes should be isolated from change by depending on interfaces instead of concrete types
* The public interface of a class should reflect its behavior, not its state
  + This implies accessor method should be minimal, and behaviorally driven

Methods within a class must also be designed with maintainability in mind.

* Methods should be small – as small as possible
  + Most code within a block can (and should) be put into its own method
  + Small methods help document code
* Each method should "do one thing"
  + When methods do one thing, they are small by nature
* Each method should correspond to one level of abstraction
  + Code should be able to read top-down, with each function call referring to a slightly more concrete idea.
* Favor polymorphism over "switch" statements
* Minimize the number of method arguments: 0 > 1 > 2 >>> 3+
  + Create "method objects" and "argument objects" as an alternative
  + Use default and keyword arguments when possible
  + Instead of using "flags" as parameters, make one function for each flag value
* Prefer returning values to modifying arguments
* Avoid side effects – they are lies
* Practice Command Query Separation
  + Functions should either do something or answer something, but not both
* Favor exceptions over returning error codes
  + Error codes must be handled immediately, but exceptions allow you separate the behavior from the error handling (and remember that error handling is one "thing")

Identify the boundaries of your system and ensure that there is a clean separation between your code and any third-party code.

* Provide boundary interfaces (wrappers) around third-party code
* Spend time to learn how to use third-party code before integration
  + Learning Tests are an excellent way to do this, and they let you check for backwards compatibility issues when updating libraries
* Define interfaces for code that does not yet exist
  + This can later be used as a wrapper, or, as a design aid for the new code

Ensure that you maintain a comprehensive test suite to ensure that future changes work as intended and do not breaking existing code.

## **Design for Object Persistence**

Persistence is the concept of maintaining the state of an object when the system is no longer running. Uses of object persistence include saving the state of a system to offline storage or transmitting the state of a system over a network (for instance, if your system leverages distributed computations).

The process of deconstructing and encoding an object is called serialization. The process of transmitting serialized data over a connection is called marshaling.

Common formats for storing serializing data include:

* Storing data in a "flat" file (e.g. plain text or binary)
* Storing data in a hierarchical format, such as XML or JSON
* Storing data in a Relational or NoSQL Database

Because anyone who intends to use the serialized data must first decode it, it is critical to design your system with object serialization in mind; if you do not, then 1) users must have knowledge about the internal structure of the class in order to serialize it themselves, and 2) every user may employ a different encoding/decoding scheme.

An alternative to traditional serialization is to cleanly separate data from behaviors within a class by way of an interface and one or more data objects. So long as a class implements the interface, only the data objects need to be stored or transmitted.

* Essentially you define an "interface" object that specifies all the behaviors, and data objects that store the data used within those behaviors. Your class then implements the interface and holds references to the data objects as components. You can then go a step further and allow the data objects to be passed in during construction.

## **What You Should Know**

Concepts

* How can you identify the public interface for a class?
* How does a constructor work?
* What makes a good constructor?
* How can you incorporate error handling into a class?
* What is unit testing?
* How do you design with reuse in mind?
* How do you design with extensibility in mind?
* How do you design with maintainability in mind?
* How do you design for object persistence?

Syntax

* How do you create a unit test using JUnit and run the test using Maven?

# 06 - Designing with Objects

## **Design Guidelines**

Refer to the course module on Agile, Scrum, and Kanban for information about modern software construction methodologies.

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

The Unified Modeling Language is the most widely used system for describing software models. Refer to the following external sources for detailed information on the topic:

* [https://www.linkedin.com/learning/software-design-modeling-with-uml (Links to an external site.)](https://www.linkedin.com/learning/software-design-modeling-with-uml)
* [https://www.uml-diagrams.org/ (Links to an external site.)](https://www.uml-diagrams.org/)
* [https://www.uml.org/what-is-uml.htm (Links to an external site.)](https://www.uml.org/what-is-uml.htm)
* [https://www.omg.org/spec/UML/2.5.1/PDF (Links to an external site.)](https://www.omg.org/spec/UML/2.5.1/PDF)
* Search for "uml tutorial" on a search engine or YouTube

UML is a very nuanced language, but most people in industry only employ the most basic elements. For this course, you are expected to know how to create a class diagram and a sequence diagram.

## **Object Wrappers**

Not every piece of software is object-oriented. Furthermore, not every piece of object-oriented software is well written. As you progress through your computer science career, you will eventually stumble across structured code that should be object-oriented, but for one reason or another is not.

* Structured code is simply a sequence of assignment, selection or iteration statements (e.g. basic C programs).

When these situations occur, one of the easiest ways to begin refactoring is to construct an object wrapper (i.e., move code into a method, and the method into a [new or existing] class).

* Wrapping structured code is especially valuable when attempting to upgrade legacy code or create unit tests.

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.

* Isolating hyper-specific sections of code makes it easy to replace those sections in the future.

## **What You Should Know**

* How can you use Scrum and Kanban to structure software development?
* How do you create a UML class diagram?
* How do you create a UML sequence diagram?
* What is the purpose of an object wrapper?

# 07 - Mastering Inheritance

## **Inheritance**

Inheritance models an "is-a" relationship between two classes and facilitates reuse by "inheriting" a subset of attributes and behaviors from one or more superclasses.

* All non-private variables and methods are accessible within a subclass.
* A subclass can override the methods of its superclass to specialize their behavior.

#### **How do you identify a potential inheritance relationship?**

One approach to using inheritance is to start out with a concrete class. When a new class appears to duplicate the behavior of an existing class, a superclass is created to link the two similar classes. All of the common attributes and methods are then moved into the superclass. If the result is an object hierarchy in which the subclasses appear to maintain an "is-a" relationship with the parent class, then the structure is kept. This process can be repeated until all such relationships have been identified.

* As an example, consider a payroll system that has to distinguish between managers, hourly employees and commission employees.

Another approach is to try and front-load the design. In the context of your problem, what sort of objects might be useful, and what sort of hierarchy between those objects might exist?

* The problem with this approach is that you are likely to create a number of classes that are not necessary and will never actually be used. Again, consider a payroll system. You might also want to include interns, seasonal workers, out-of-state employees, foreign employees, volunteers, etc.; but, what is the likelihood that you'll actually need all of those subtypes?

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.

The following is an example of a UML class diagram showing an inheritance relationship between objects:

@startuml  
class Employee  
  
class Manager  
class HourlyEmployee  
class CommissionEmployee  
  
Employee <|-- Manager  
Employee <|-- HourlyEmployee  
Employee <|-- CommissionEmployee  
@enduml

## **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?

* Most commonly, this is done when there is no good default implementation. Abstract classes are intended to be used as superclasses within an inheritance hierarchy, and there is an assumption that the subclasses are a better place to implement the method.

Why use an inheritance hierarchy with an abstract class instead of just defining the methods in the subclasses and forgoing inheritance?

* Polymorphism. Subclasses of an abstract class should still abide by a strict "is-a" relationship. Additionally, because abstract classes may still contain concrete methods and attributes, those are also accessible to the subclasses.

More generally, abstract classes allow developers to provide a contract to users of the class. This contract ensures that any subclass must implement the abstract methods, and will therefor support the associated behaviors. This allows developers to use the abstract class as a type within their own code, but leverage polymorphism to run a user's code during execution.

* This is how many frameworks and third-party APIs work. You pass your own objects to the library code, and the behavior changes accordingly.

What if a subclass does not abide by a strict "is-a" relationship, but it should still abide by the contact? Or what if a class should satisfy multiple contracts?

* In this case, an interface is the correct solution.

## **Interfaces**

The term interface is an overloaded term in computer science. In the context of designing an object-oriented system, 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 following example illustrates how to design with interfaces:

@startuml  
interface Nameable {  
    getName()  
    setName()  
}  
  
interface Trainable {  
    sit()  
    stay()  
    rollover()  
}  
  
class Mammal {  
    eat()  
    breath()  
    reproduce()  
    die()  
}  
  
class Dog  
class Car  
class Planet  
  
Nameable <|.. Mammal  
Nameable <|.. Car  
Nameable <|.. Planet  
  
Trainable <|.. Dog  
  
Mammal <|-- Dog  
@enduml

In this example, the relationship between `Nameable` and `Mammal`, `Car`, and `Planet` is not a strict "is-a" relationship. Nor is the relationship between `Trainable and `Dog`. However, it is fair to say that a Dog "is-a" `Mammal`.

So if an interface does not model an "is-a" relationship, what does it model?

* Interfaces are used to model "behaviors-like-a" relationships.

## **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.
  + Consider a Bird object hierarchy. If the base class has a "fly" method, then so do all the subclasses. But, do all birds fly? No.
  + Consider a square and a rectangle. Is a square a special type of rectangle? No; a square only has one size, not a length and width.

You must also ensure that your unit tests verify the behavior of your system when a subclass is substituted in for the superclass. If the system does not respond as expected, it is an indication that you have violated the Liskov Substitution Principle and your subclass does not have a true "is-a" relationship with the superclass. In this situation, it may be best to refactor the subclass to either make use of interfaces or into a stand-alone class that leverages composition as necessary (or both).

## **What You Should Know**

Concepts

* What is inheritance?
* When should you use inheritance?
* What is an abstract class?
* When should you use an abstract class?
* What is an interface?
* When should you use an interface?

Syntax

* How can you convert a simple class diagram into Java code?
* How do you define an abstract class?
* How do you define an interface?
* How do you implement inheritance with a concrete class?
* How do you implement inheritance with an abstract class?
* How do you implement inheritance with an interface?
* How do you create unit tests for polymorphic code?

# 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.

Unlike inheritance, composition does not silently modify the structure of an object - any desire to use the behaviors of a component must be made explicit. The following code example illustrates this concept.

*// Engine.java*

**public** **class** **Engine** {

**private** int velocity;

**public** Engine() {

velocity = 0;

}

**public** void drive() {

velocity = 10;

}

**public** void brake() {

velocity = 0;

}

}

*// Car.java*

**public** **class** **Car** {

**private** **Engine engine;**

**public** Car() {

engine = **new** Engine();

}

**public** void drive() {

engine.drive();

}

**public** void brake() {

engine.brake();

}

}

Here we see that the `**Car` creates an `Engine**` object and then **delegates** some of its behavior to that `Engine` object through explicit method calls. If we had designed the `Car` class to inherit from `Engine`, then we would not need to store a reference to an `Engine`, nor would we have defined the `drive` and `break` methods inside of `Car`.

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.

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 following example illustrates these three different notions.

@startuml  
class Road {  
    - friction: double  
    + getFriction(): double  
}  
  
class Car {  
    - engine: Engine  
    - wheels: ArrayList<Wheel>  
   - passengers: ArrayList<Passengar>  
  
    + Car(Passenger): Car  
    drive(Road)  
    brake(Road)  
}  
  
class Engine {  
    drive(Road, ArrayList<Wheel>)  
    brake(Road, ArrayList<Wheel>)  
}  
  
class Wheel {  
    - radius: double  
    + getRadius()  
}  
  
Road - Engine  
  
Car "1" \*-- "1" Engine  
Car "1" \*-- "4" Wheel  
Car "0..n" -- "1" Road  
**Car "1" o-- "1..n" Passanger**  
Engine - Wheel  
@enduml

Here we see that a car is composed of 1 `Engine` instance and multiple `Wheel` instances, and uses aggregation to model the passengers. It has an association with `Road` through the drive and break methods. Furthermore, Engine has an association with `Road` and `Wheel` through its own `drive` and `break` methods.

* It is common for developers to hide most of the association links in a diagram, as their inclusion can often lead to diagrams that are difficult to read.
* The numbers associated with each line represent the cardinality of the relationship: how many instances each object is associated with.
  + Specifying cardinality in a model can help guide the implementation by giving insight into some of the class's constraints

#### **How do you identify potential composition relationships?**

You can use the same approaching as when looking for inheritance. Start off with a monolithic class and break off components as the class begins to take on more than one responsibility, or consult the design documentation to get an initial picture of which components may need to be created and where those components will be used.

## **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 (direct).

* Concepts such as dependency injection, where instances of components are passed in as parameters, can help reduce the impact of future changes.

## **Dependency Injection**

The following example illustrates how an initial design may give rise to a solution that makes use of dependency injection.

Suppose that we want to model some animals for a Zoo application. Our first idea might be that we can make an object hierarchy out of the animals.

**class** **Mammal** {

**public** String eat() {

**return** "eating";

}

}

**class** **Bat** **extends** Mammal {

**public** String fly() {

**return** "flying";

}

}

**class** **Dog** **extends** Mammal {

**public** String walk() {

**return** "walking";

}

}

**public** **class** **AppTest** {

@Test

**public** void test\_animals() {

Dog fido = **new** Dog();

assertEquals(fido.eat(), "eating");

assertEquals(fido.walk(), "walking");

Bat brown = **new** Bat();

assertEquals(brown.eat(), "eating");

assertEquals(brown.fly(), "flying");

}

}

The problem with this approach is that we're having to put specific behavior into each of the subclasses, since not all animals fly and not all animals walk. What would we do about an animal that can both walk and fly, such as a bird? Do all birds fly?

One alternative would be to take advantage of interfaces and allow a dog to implement a Walking interface. In the case of a bird, it might implement a Walking and Flying interface, but this too has flaws - every class would need to provide its own behavior, even when it makes more sense to reuse existing behavior.

What if we used composition?

**class** **EatingBehavior** {

**public** String eat() {

**return** "eating";

}

}

**class** **FlyingBehavior** {

**public** String fly() {

**return** "flying";

}

}

**class** **WalkingBehavior** {

**public** String walk() {

**return** "walking";

}

}

**class** **Bat** {

EatingBehavior eatingBehavior = **new** EatingBehavior();

FlyingBehavior flyingBehavior = **new** FlyingBehavior();

**public** String eat() {

**return** eatingBehavior.eat();

}

**public** String fly() {

**return** flyingBehavior.fly();

}

}

**class** **Dog** {

EatingBehavior eatingBehavior = **new** EatingBehavior();

WalkingBehavior walkingBehavior = **new** WalkingBehavior();

**public** String eat() {

**return** eatingBehavior.eat();

}

**public** String walk() {

**return** walkingBehavior.walk();

}

}

**public** **class** **AppTest** {

@Test

**public** void test\_animals() {

Dog fido = **new** Dog();

assertEquals(fido.eat(), "eating");

assertEquals(fido.walk(), "walking");

Bat brown = **new** Bat();

assertEquals(brown.eat(), "eating");

assertEquals(brown.fly(), "flying");

}

}

 This solution appears to be a bit more flexible. We can now create animals by composing different behavioral elements. The main problem with this approach is that we've hard-coded the specific behaviors to be used. The `WalkingBehavior` of a `Dog` will always be `WalkingBehavior`, even if someone else develops a new subclass they'd rather use. (We could change it, but then we'd have to recompile the class.)

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

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

**private** EatingBehavior eatingBehavior;

**private** FlyingBehavior flyingBehavior;

**public** Bat() {

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

}

**public** Bat(EatingBehavior eatingBehavior) {

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

}

**public** Bat(FlyingBehavior flyingBehavior) {

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

}

**public** Bat(EatingBehavior eatingBehavior, FlyingBehavior flyingBehavior) {

**this**.eatingBehavior = eatingBehavior;

**this**.flyingBehavior = flyingBehavior;

}

**public** void setEatingBehavior(EatingBehavior eatingBehavior) {

**this**.eatingBehavior = eatingBehavior;

}

**public** void setFlyingBehavior(FlyingBehavior flyingBehavior) {

**this**.flyingBehavior = flyingBehavior;

}

**public** String eat() {

**return** eatingBehavior.eat();

}

**public** String fly() {

**return** flyingBehavior.fly();

}

}

 At this point, our class is fairly robust and extensible. The only other potential issue is that `eat()` and `fly()` are optional. To "fix" this, we could use interfaces to create a set of contracts the the `Bat` must implement - but this may often be overkill.

## **What You Should Know**

Concepts

* What is composition?
* When should you use composition?
* What is dependency injection?

Syntax

* How do you implement composition?
* How do you implement dependency injection?