Encapsulation & OOP

- **Non Object-Oriented Approach:**
  - All functions are static
  - Variables are all either declared locally or passed in as parameters.

  ⇒ Unorganized, not scalable, and not modular.
- **Objected-Oriented Programming:** A programming paradigm that organizes software design around objects rather than functions or logic. Objects are instances of classes, which are blueprints that define the properties (attributes) and behaviors (methods) of the objects. OOP promotes principles such as encapsulation, inheritance, and polymorphism, which help in creating modular, reusable, and maintainable code. By modeling real-world entities as objects, OOP allows developers to create complex systems that are easier to understand, modify, and extend.
  - Formalizes the relationship between information and functionality to create a metaphor (abstraction).
  - This abstraction:
    - Provides code organization.
    - Makes our code easier to read.
    - Protects our data from being used incorrectly.
- **Creating a Well-Formatted Object:**
  1. Create a new file
  2. Name the object (class name)
  3. Declare instance fields at the top of the class
  4. Create your constructor:
     - Constructor is a special type of method whose job is to create and initialize a new instance.
     - It is called whenever the `new` keyword is used.

    - E.g. `Dragon smaug = new Dragon();`
- Declaration:
    - Its name must match its class name.
    - Does not have any sort of return value in its signature.
- Within the constructor, the keyword `this` refers to the instance fields of the current object.
- Any information needed should be passed in as parameters.
    - Code in the constructor is responsible for making sure that the fields of `this` are appropriately set.
- An object can have multiple constructors.

5. Define instance methods:
    - These are functions that depend on a specific instance of the current object.
    - Declare instance methods without the `static` keyword.
        - With the `static` keyword, they are not instance methods anymore.
    - Instance methods only make sense in the context of a specific instance.
    - Within an instance method, the keyword `this` provides a reference to the current object itself.
        - E.g. `this.power`
6. Access Modifiers:
    a. `public`: The current field or method is accessible from any other class, both inside and outside the package in which it is defined.
    b. `private:` The current field or method is accessible only within the class in which it is defined. It cannot be accessed from outside this class.
    c. `protected`: The current field or method is accessible within the same package and by subclasses, even if they are in different packages.

d. **default** (**no modifier**): The current field or method is accessible only within the same package. It is not accessible from outside the package, even by subclasses. This is also known as "**package-private**."

7. Define `toString()`:
    a. `toString()`: An `@Override` method from the object class.
    b. By default, it prints the memory location of the current object.
    c. It is called whenever `System.out.print` is called.
    d. Must follow this definition:

```
@Override
public String toString() {
    return this.name + ": proper string here";
}
```

Example Encapsulation:

```
public class Dragon {
    int power;
    int toughness;

    public Dragon() {
        this.power = 100;
        this.toughness = 100;
    }

    public Dragon(int power, int toughness) {
        this.power = power;
        this.toughness = toughness;
    }

    public void defend(int power){
```

```
            this.power += power;
        }

    }
```

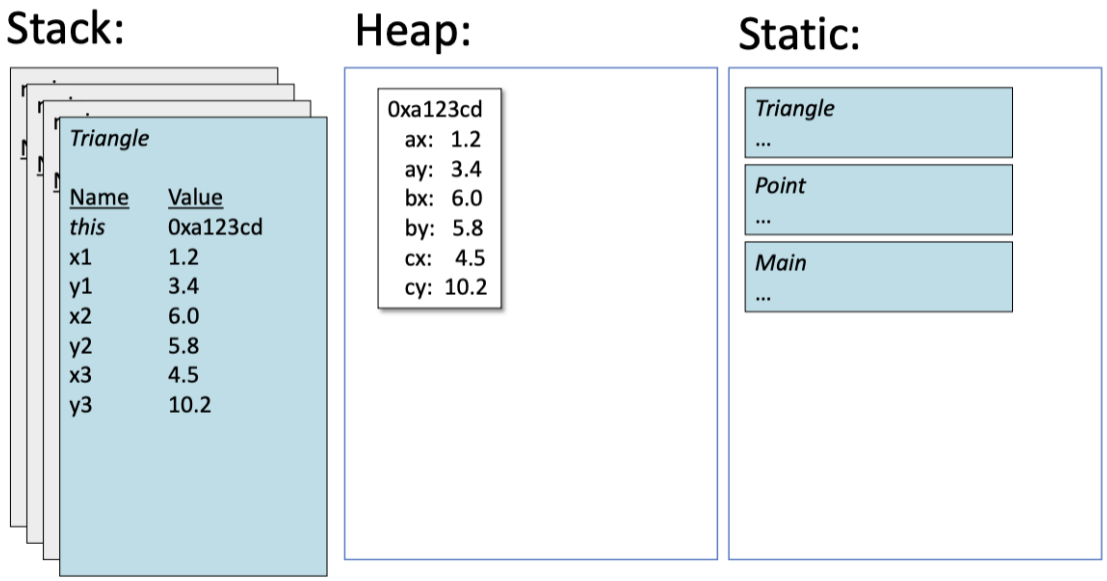
- **The Three Pillars of Object-Oriented Programming:**
    - **Encapsulation**: Encapsulation is the concept of bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class.
        - It also involves restricting access to certain details of an object, exposing only what is necessary through public methods, while keeping the internal implementation hidden.
        - This promotes modularity and helps protect the integrity of the object's data by preventing unauthorized access or modification.
        - Objects often represent some real-world concept, e.g., a person, or a bank account and contain:
            - Fields aka Instance Variables: object-specific data: e.g., a person's name.
            - Methods: procedures that act upon the fields: e.g., changing a person's name.
        - Use getters and setters when absolutely necessary (i.e. Getters and setters should be used thoughtfully and only when they serve a clear purpose, such as enforcing encapsulation, controlling access, or maintaining flexibility in your code. Overusing them can lead to poor design, increased complexity, and tighter coupling between classes, which are contrary to the principles of good object-oriented design.)
    - **Inheritance**:
        - Inheritance is the mechanism by which one class (the child or subclass) can inherit properties and behaviors (attributes and methods) from another class (the parent or superclass).

        - This allows for code reuse, as common functionality can be defined in a base class and shared with multiple derived classes.
        - Inheritance also helps in creating hierarchical relationships between classes, where more specialized classes extend the functionality of more general ones.
    - **Polymorphism**:
        - Polymorphism allows objects of different classes to be treated as objects of a common super class.
        - It is the ability to define methods in a base class and override them in derived classes, allowing for different implementations while sharing a common interface.
        - Polymorphism enables the same method or operation to behave differently on different objects, depending on their actual class, promoting flexibility and extensibility in code design.
- **Principles of Encapsulation:**
    - **Principle 1**: **Shield object internals from the rest of the program:**
        - This prevents instance fields from accidentally being changed or intentionally abused.
        - Allows internal code to be refactored without breaking external code.

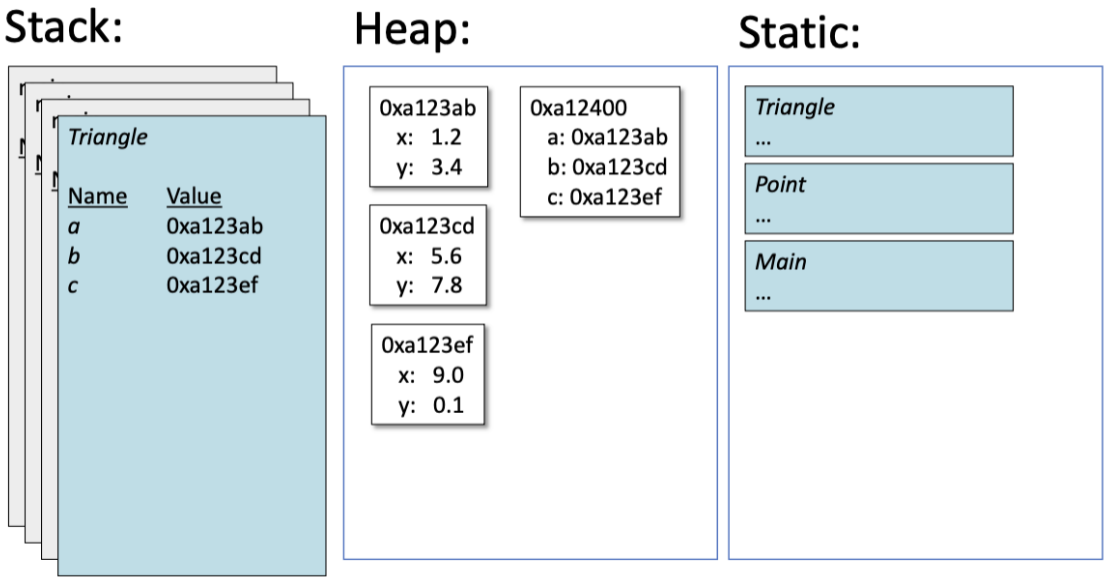
      ⇒ Use the `private` access modifier.
        - Instance fields must be private, with public `getter` methods defined for the encapsulated fields.
        - No redundant fields.
    - **Principle 2**: **Explicitly define "external" and "internal" behavior:**
        - Makes code more modular.
        - Makes objects easier to understand, maintain, use, and change.

      ⇒ Use an `interface` .
- **Object Construction in Stack, Heap, and Static memories:**

- In the Stack:
  - Stack memory is used for static memory allocation, including method calls, local variables, and control flow data. Objects themselves are not stored on the stack, but references to objects can be. When a method is invoked, its local variables are allocated on the stack, and when the method exits, these variables are automatically deallocated.
- In the Heap:
  - The heap is used for dynamic memory allocation. When you use new to create an object, the object is allocated on the heap, and memory is managed manually (through garbage collection in languages like Java). Objects stored in the heap are accessible throughout the application, as long as there are references pointing to them.
- In Static Memory:
  - Static memory is allocated once when the program starts and is used for static variables and data that persist throughout the program's lifetime. Static memory is not deallocated until the program exits. It stores static fields, global variables, and constants defined in classes.
- E.g. For `DoubleTriangle t = new DoubleTriangle(x1, y1, x2, y2, x3, y3);` :
  - **Stack:** The reference variable t is stored on the stack.
  - **Heap:** The DoubleTriangle object, along with its properties (like coordinates or points), is allocated in the heap memory.
  - **Static Memory:** If DoubleTriangle has static fields, those fields are stored in the static memory.



- E.g. For `PointTriangle t = new PointTriangle(a, b, c);`:
  - **Stack:** The reference variable t is on the stack.
  - **Heap:** The PointTriangle object is created in the heap. The points a, b, and c are also typically stored in the heap if they are objects.
  - **Static Memory:** Any static fields in PointTriangle are stored in static memory.

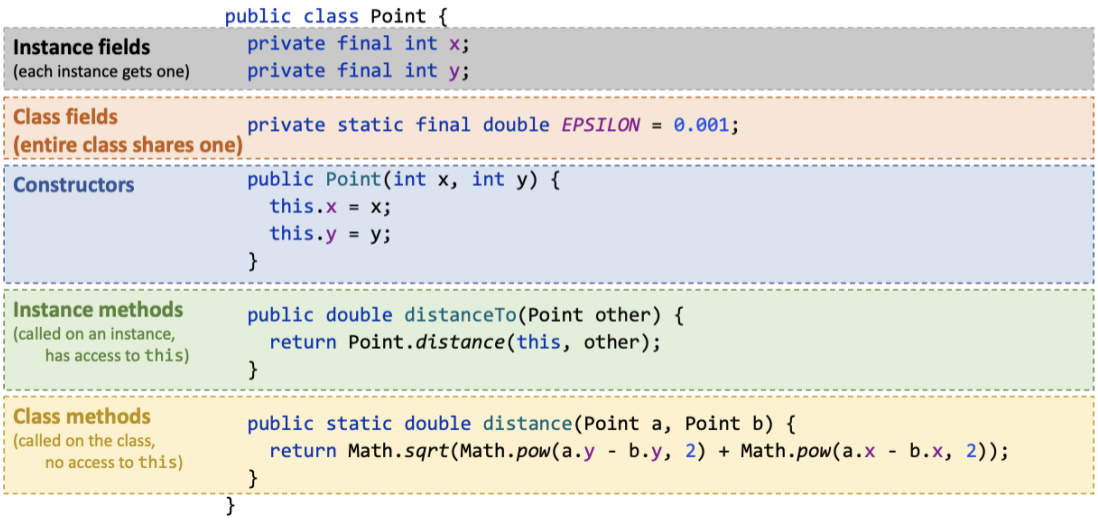


- **Primitives vs Objects:**
  - Primitives:

        - Lower-case types
        - Represented as binary values
        - Are passed by value
    - Objects:
        - Types are all uppercase
        - Inherit from the Object class
        - Passed by memory pointer on the heap
- **Immutable Objects:**
    - **Immutable**: A computer science word describing a value that cannot be changed:
        - If a value cannot change after it is initialized, we say that it is immutable.
        - `final` keyword in Java.
        - If an object's fields are immutable, we say that the object is immutable.

      ⇒ Immutable objects are easy to reason about because once one is created, its internal field values stay the same forever.
- **Choosing Fields - Rule of Thumb**: When designing a class, choose to include fields that fundamentally identify the object and try to avoid redundant fields.
- **Static Class Fields and Methods:**
    - Instance fields are data associated with each instance.
        - Every object has its own set of values.
    - Class fields are data associated with the class as a whole.
        - Declared with the `static` keyword.
        - Accessible via the class name or directly if being used in the class itself.
        - **Naming Convention**: All capitalized.
        - **Best Practices**:

        - Initialize the static fields when declared.
        - Declared with the `final` keyword to indicate that it won't ever change.
    - Class methods are functions related to the abstraction but not specific to any instance.
        - Has access to the `this` object.
    - Example:



- **Derived Fields:**
    - A derived field is an imaginary "field" that is actually just a combination or transformation of other fields.
    - Derived fields do not need to be stored, because they can be calculated on-demand inside a getter method.
    - E.g.

```
public double getArea() {
    return side * side;
}
// Field 'area' is not necessary, side * side is the
// derived field
```

- **Setter Methods:**
  - When writing a setter method, we must add code to validate the incoming value *before updating the field*.
  - Setter methods must always return `void`.
  - If an improper value is detected, **throw an error** to end the program.
  - E.g.

```
public void setSide(double side) {
    if (side <= 0) {
        throw new IllegalArgumentException("Side cannot be
    }
    this.side = side;
}
```

- **JavaDoc Commenting:**
  - **Tags List:**
    - `@author`: Describes an author.
    - `@param`: Provides information about method parameters or the input it takes.
    - `@see`: Generates a link to other element(s) of the document.
    - `@version`: Provides the version of the class, interface, or enum.
    - `@return`: Provides the return value.
  - **Format**:

```
/**
 *
 * JavaDoc comment
 *
 */
```