Chapter 4: Writing Classes

4.1 Classes and Objects Revisited

A **class** defines what an object will look like and how it will behave. An **object** is created from a class and stores its own data in instance variables that can be modified over time.

Objects have **state**, defined by the **attributes** associated with the object.

Objects have **behaviours**, defined by the **operations** associated with the object.

4.2 Anatomy of a Class

Every class can contain **data declarations** and **method declarations.** Collectively, these are called the **members** of a class.

data declarations represent the data that will be stored in each object of the class.

method declarations define the services that those objects will provide.

Special Methods:

Constructor - A special method that initializes a new object of the class. It has the same name as the class and is automatically called when an object is created. Constructors establish the initial state of an object by setting its instance variables.

Overrides (eg `toString()`) - Methods that override inherited methods from the Object class, such as toString(), which returns a string representation of the object. Overriding these methods allows you to customize how your objects behave when used in common operations.

The `toString()` method converts an object to its string representation. When overridden, it lets you define how your object should be displayed when printed.

```
@Override

public String toString() {

    return "Student: " + firstName + " " + lastName;

}
```

Instance Data

Constant and variables can be declared inside a class but not inside any method. The location a variable is declared defines its **scope** (The area in the class from which it can be referenced)

Variables declared outside the scope of a method are called **instance data**, because new memory space is reserved for the variable each time an object is instantiated. Instance data can be accessed from anywhere inside the class (and outside if public).

UML Class Diagrams

UML (Unified Modeling Language) is the most popular notation for representing the design of an object-oriented program. They help us visualize the contents and relationships among the classes of a program.

In UML, each class is represented as a rectangle containing sections for the class name, data and methods.

The types listed after each method is its return type.

The arrow in this diagram indicates a relationship between the classes. In this case, a dotted arrow indicates one class *uses* the other.

4.3 Encapsulation

Objects should be *self governing*. This means that the instance data of an object should be modified only by that object. This prevents inappropriate and difficult to predict interaction with a class. This concept is called **encapsulation**.

Properly encapsulated objects only are interacted with through a specific set of methods that define the services it provides. These methods define the **interface** between the object and the program that uses.

We accomplish encapsulation in Java using **modifiers** - reserved words that specify a characteristic of a programming language construct.

An example is `final` which is used to declare a constant.

Visibility Modifiers

Visibility modifiers control access to members of a class.

The three main visibility modifiers in Java are:

- `public` - Members are accessible from any class
- `private` - Members are only accessible within the declaring class
- `protected` - Members are accessible within the declaring class and its subclasses
- **default** - If no visibility modifier is specified, the member has package-level access, meaning it's accessible only to classes in the same package.

Instance variables should always be `private` to respect encapsulation.

Local variables cannot be assigned visibility modifiers because they are already inaccessible from outside the method.

Constants may be made `public` if necessary, as they cannot be changed.

Methods may or may not be public depending on their purpose:

- **Service methods** are methods that provide services to other classes in the program. They must be `public`.
- **Support methods** are Methods that support the service methods in the class, but are not themselves need as services. They should generally be `private` .

Accessors and Mutators

accessors and mutators provide controlled access to private instance data, maintaining encapsulation while allowing necessary interactions with the object's state.

An **accessor method** (also called a getter) allows other objects to obtain the value of an instance variable. They typically start with "get" and return the value of a specific instance variable.

```
private int age;

public int getAge() {

   return age;

}
```

A **mutator method** (also called a setter) allows other objects to modify the value of an instance variable. They typically start with "set" and accept a parameter that will become the new value of the instance variable. Setters should ensure that only valid values are permitted to modify instance variables.

```
public void setAge(int newAge) {

                 if (age > 0){

                    age = newAge;

                 }

}
```

4.4 Anatomy of a Method

A **method** is a named group of programming statements that is part of a class.

When a method is called, the **flow of control** jumps to the method, executes the method sequentially, and then returns to the calling point.

A *method declaration* consists of several key components:

Header - The first line of a method declaration includes:

- Visibility modifier (public, private, protected)
- Return type (or void if no return value)
- Method name
- Parameter list in parentheses (can be empty)
    - The parameters defined in the method declaration are called **formal parameters.** They are used as variables in the method body.
    - In an invocation, the actual parameters declared in the method call are called **arguments,** and are copied into the formal parameters.

Body - The code block enclosed in curly braces that contains:

- Local variable declarations
    - Variables declared inside a method are *local data* to that method. That means that they cannot be accessed outside of that method.
    - Because local data and instance (class level) data operate at different scopes, it is permitted to have a local variable with the same name as an instance variable. This can be confusing however, and so it best avoided.
- Statements and expressions
- Return statement (Unless return type is `void`)
    - a Return statement uses the reserved work `return` and terminates the method execution by returning the provided value.
    - It is good practice not to use more than one `return` statement in a single method unless absolutely necessary.

```
public double calculateArea(double length, double width) {

   // Local variable declaration

   double area;



   // Method logic

   area = length * width;



   // Return statement

   return area;
```

}

The **parameter list** defines any input values the method needs to perform its task. Each parameter includes both its type and name.

The **return type** specifies what kind of value (if any) the method will send back to the code that called it. If a method doesn't return anything, its return type is declared as `void`.

Methods should be designed to perform a single, well-defined task, following the principle of single responsibility.

Invoking (Calling) a Method

When invoking a method, we use the method name followed by parentheses containing any required arguments. The arguments must match the parameter types specified in the method declaration.

When a method is called from outside the class, it uses dot notation `object.methodName()`

When a method is called from within a class, it is referenced directly `methodName()`

Here's a basic example of calling a method:

// Calling a method on an object

Rectangle box = new Rectangle();

double area = box.calculateArea(5.0, 3.0);

// Calling a method within the same class

printArea(area);

4.5 Constructors Revisited

A **constructor** is a special method invoked when an object is instantiated. It's used to set up the class and initialize object variables.

A constructor is different from a regular method in two ways:

- Its name must match the class name (e.g., Die class has Die constructor)
- It cannot have a return type specified in the header (not even void)

If no constructor is defined, Java provides a default no-parameter constructor. This default constructor doesn't modify the new object.

It is permitted to have multiple constructors if needed. For example:

```
public class Die {

                  public static final int MAX = 6;

   private int faceValue;



   // Parameter-less constructor

   public Die() {

      faceValue = 1;

   }

   //constructor with a parameter.

                  public Die(int value){

      if (value >= 1 && value <= MAX) {

         faceValue = value;

      } else {

         faceValue = 1;  // Default to 1 if invalid value

      }

   }

}
```

4.L Records

(from Bruce's lecture slides)

A **record** is a special type of class in Java that provides a compact way to create immutable data-holding classes.

A record should be identifiable by its data.

Records have **immutable** state - their data cannot be modified after creation.

Declaring Records

A record class can be declared in one line:

```
record Point(double x, double y) {}
```

Features of a Record

When a record class is declared, it automatically generates:

- private final instance variables for each formal parameter
- public accessor methods for each private variable
- Implementations of toString(), equals(), hashCode() methods

It is also possible to declare additional methods within a Record.

When to Use Records

- For immutable data structures
- When data should be transparently accessible
- To signal design intent to compiler and developers. this helps with:
  - Compiler optimizations
  - Better concurrency handling