Method Overloading

1. **Online Shopping Cart** 🛒 Create a class called `ShoppingCart` that manages items. It should have:

- **Constructors**: One constructor that initializes the cart with a default capacity of 10 items. Another constructor should accept an integer `capacity` to set the initial size. A third constructor should accept an existing array of `Item` objects to pre-populate the cart.
- **Methods**: An `addItem` method that is overloaded. One version should accept an `Item` object. Another should accept `String name`, `double price`, and `int quantity` to create and add a new `Item`. A third version could accept a `String name` and `double price` and default the quantity to 1. The class should also have a `calculateTotal` method that returns the total cost of all items.

The calculations would involve summing up the price of each item multiplied by its quantity to get the total.

---

2. **Fitness Tracker** 🏃 Design a class named `FitnessTracker` to log various workout activities. It should have:

- **Constructors**: A default constructor that initializes a tracker with a specific activity (e.g., "Running"), a duration of 30 minutes, and a calorie count of 250. A second constructor should accept the activity name, duration in minutes, and calories burned. A third constructor should accept the activity name and duration and calculate the calories based on a predefined formula (e.g., calories = duration * 8).
- **Methods**: An overloaded `logActivity` method. One version could accept a `String activityName` and `int duration`. Another could accept `String activityName`, `int duration`, and `double distanceInMiles`. A third could accept all three plus the `int caloriesBurned`. The method should update the tracker's total duration and calories.

The calculations would involve calculating calories based on a formula and summing up total duration and calories across different logged activities.

---

3. **Complex Number Operations** ➕ Create a class named `ComplexNumber` to represent and perform operations on complex numbers. A complex number is of the form $a + bi$. It should have:

- **Constructors**: A constructor that accepts two `double` values for the real and imaginary parts. A second constructor that accepts only one `double` value, treating it as the real part and the imaginary part as 0. A default constructor that initializes both parts to 0.
- **Methods**: Overload a method called `add`. One version should accept another `ComplexNumber` object and return a new `ComplexNumber` that is the sum of the two. Another version could accept a `double` and add it to the real part of the current complex number. A third version could accept two `double` values and add them to the real and imaginary parts respectively. The `add` method should correctly perform the addition $(a+bi) + (c+di) = (a+c) + (b+d)i$.

The calculations involve the basic arithmetic of complex numbers.

---

4. **Library Management System** 📚 Develop a `LibraryItem` class to manage different types of items in a library. It should have:

- **Constructors**: A constructor for a `Book` that takes the title, author, and ISBN. A constructor for a `DVD` that takes the title, director, and duration. Another constructor for a `Magazine` that takes the title, publisher, and issue number. All constructors should also accept a unique item ID.
- **Methods**: An overloaded method `checkOut`. One version should accept a `String memberId` and a `Date dueDate` for a standard checkout. A second version could accept a `String memberId` and automatically set the due date to 14 days from the current date. A third version could accept the `memberId` and a `String checkoutType` (e.g., "short-term", "long-term"), which sets the due date accordingly (e.g., 7 days for short-term, 28 for long-term).

The problem involves calculating new dates based on a set duration.

---

5. **Invoice Generator** 🧾 Create a class called `Invoice` to generate invoices for a business. It should have:

- **Constructors**: A constructor that accepts the customer name, invoice number, and an array of `InvoiceItem` objects. A second constructor that accepts just the customer name and invoice number, initializing an empty list of items. A third constructor could accept the customer name, automatically generate an invoice number, and initialize an empty list of items.
- **Methods**: An overloaded `addItem` method. One version adds a `String itemName`, `double unitPrice`, and `int quantity`. A second version adds an `InvoiceItem` object directly. A third version adds an item name and price and defaults the quantity to 1. The class should also have a `calculateTotal` method that returns the subtotal, tax amount, and the final total as an object or a formatted string.

The calculations would involve calculating the subtotal (sum of all items), then the tax amount based on the subtotal (e.g., 7%), and finally the grand total.