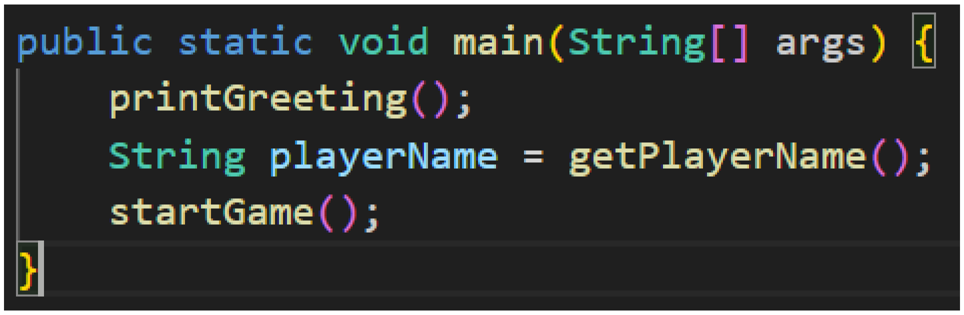
Java Best Practices!

Good, clean code must follow certain conventions in Java. Learning and using these early on will not only enhance your coding skills, but it will also prepare you for collaborative projects in which code quality and consistency matter. We'll cover some of the must-know tips below!

1. Keep it short

When writing methods in Java, it's important to keep them short and focused, as they are easier to read, understand, and maintain. Short, concise methods allow you to **isolate functionality**, making it easy to test your code and fix bugs, too. By keeping methods concise, you adhere to best practices that improve the overall quality of your code.

Here is an example of a short and focused main method:



```java
public static void main(String[] args) {
    printGreeting();
    String playerName = getPlayerName();
    startGame();
}
```

This code is considered good practice because it keeps the main method clean and focused by delegating specific tasks to separate, short methods. In this case, the method printGreeting() will greet the user, getPlayerName() will store the user's name using a Scanner, and startGame() will initiate the game. This code can be easily read, understood, and tested by other programmers.

2. Naming

Choosing the right names for variables, methods, and classes is crucial for writing good, clean Java code. Following established naming conventions not only makes your code more readable but also helps other developers understand your intentions quickly.

a. Variables

Use **camelCase** for naming. Start with a lowercase letter, then capitalize the first letter of each following word.

Examples include: customerAge, accountBalance, numCardsInHand…

Choose names that clearly indicate what the variable will represent. Do not start names with _ or $. Avoid using single letters, and certain keywords like int, float, or string.

b. Methods

Just like variables, you should be using camelCasing to name methods. Generally, these names start with verbs and describe the functionality of the method.

Examples: calculateTotalPrice(), isRegistered(), convertToBinary()

Make sure to use consistent naming conventions for similar methods!

Example: getAccountBalance() & setAccountBalance()

c. Classes

Use UpperCamelCase for naming classes (each word starts with an uppercase letter). Generally, these are nouns that represent objects or concepts.

Examples: CheckingAccount or TemperatureConverter

d. Constants

You should name constants with UPPERCASE letters with underscores separating each word. Use specific words that clearly indicate what the constant is.

Examples: MAX_USERS, OVERDRAFT_LIMIT, DEFAULT_LANGUAGE

Remember to declare your constants as **static** and **final**!

3. Test, test, and test!

Testing is a fundamental aspect of software development that ensures your programs will function correctly and efficiently.

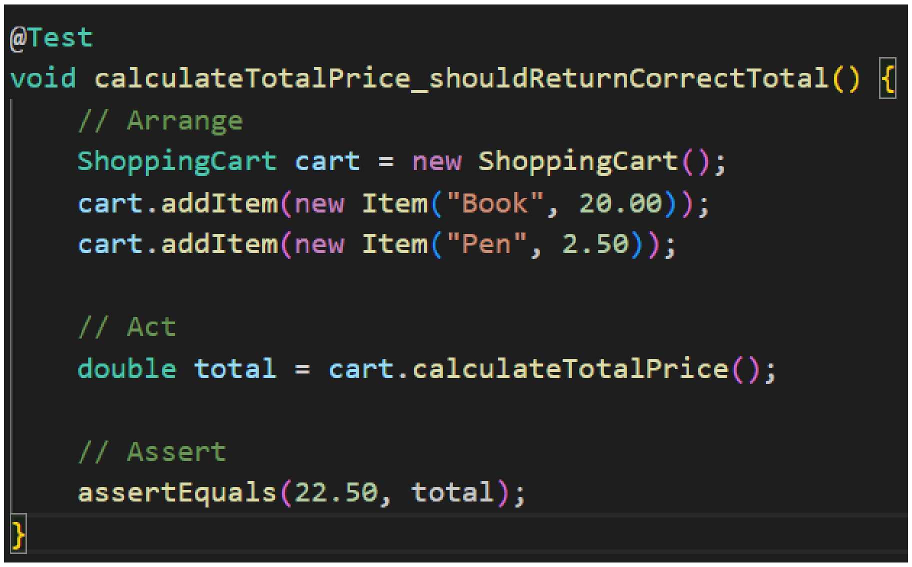
1. Make sure to test early and often!

Write unit tests to check the functionality of individual components of your program. Testing early helps you catch bugs and exceptions. After testing, each individual part of your program should work as intended, and they should be easy to make changes to if necessary.

DO NOT WAIT UNTIL THE PROGRAM IS COMPLETE TO TEST!

2. Important Testing Considerations

Make sure that you set up the testing data and environment before the assert statement.



```java
@Test
void calculateTotalPrice_shouldReturnCorrectTotal() {
    // Arrange
    ShoppingCart cart = new ShoppingCart();
    cart.addItem(new Item("Book", 20.00));
    cart.addItem(new Item("Pen", 2.50));

    // Act
    double total = cart.calculateTotalPrice();

    // Assert
    assertEquals(22.50, total);
}
```

In this example, a ShoppingCart object is created and filled with two items before testing the calculateTotalPrice() method.

Use assertEquals, assertTrue, and assertThrows over the generic assert.

Provide clear and custom failure messages, too!

Example: assertEquals(expected, actual, "Total price should match the sum of item prices");

4. Comments

Writing good comments is crucial for all programs that you will write for this and future courses. Writing clear comments will mean that your code can be easily understood by others and shortens time spent debugging.

a. **Try to explain why, not what** (your thoughts vs what the code is doing)

A comment like **//Use binary search for efficient look-up in the sorted list** is great because it explains why binary search is used, not what the code is actually doing.

b. **Avoid repeating or unnecessary comments**

Don't just state the obvious, not every line needs to be commented. When declaring a simple variable or using an assignment statement, let the code speak for itself.

c. **Write clear Java Docs**

Write javadocs for each method and class you create! Start with a brief summary of what the method or class will do / represent. Use tags like **@param** or **@return** so anyone who reads the javadoc can make things clear

```java
/**
 * Calculates the total price of items in the cart.
 * @param cartItems List of items in the cart
 * @return Total price of the items
 */
public double calculateTotalPrice(List<Item> cartItems) { ... }
```

5. Common Mistakes

1. **Long, Complicated Method**
Long methods are harder to read, test, and maintain. Break down large methods into smaller, single-purpose methods.

2. **Inconsistent Naming Conventions**
Inconsistent naming makes the code harder to read and understand. Stick to established naming conventions: camelCase for variables and methods, UpperCamelCase for classes, and UPPERCASE for constants.

3. **Not Testing Early or Often Enough**
I cannot stress this enough. Delaying testing increases the chances of undetected bugs and makes debugging harder. Write unit tests as you develop each feature. Test small parts of your program independently before integrating them.

4. **No Magic Numbers**
Hardcoded values make the code less readable and harder to update if the values need to change. Define these values as constants with descriptive names, making it easier to adjust the value in one place if needed.

5. **Lack of / Poorly Written Comments**
No comments make code harder to understand, especially for others. Redundant comments clutter the code without adding useful context. Write meaningful comments explaining the "why" behind complex logic, and use Javadoc for documenting methods and classes.