Understanding Polymorphism in Java with Selenium: A Step-by-Step Guide

Polymorphism in Java is a powerful concept that allows a single interface to be implemented by multiple classes or methods. This flexibility enhances code reusability and adaptability, making it particularly useful in test automation frameworks like Selenium.

---

1. Compile-Time Polymorphism (Method Overloading)

Definition:
Compile-time polymorphism occurs when multiple methods in the same class share the same name but differ in their parameter lists. The appropriate method is selected during the compilation phase.

Key Benefits in Selenium:

- Simplifies handling different types of inputs without creating extra method names.
- Optimizes code for varying test requirements.

Code Example 1: Overloaded Methods for Browser Actions

```java
public class BrowserActions {
    // Overloaded method 1: Open browser with a URL
    public void openBrowser(String url) {
        System.out.println("Opening browser with URL: " + url);
    }
    // Overloaded method 2: Open browser with a URL and timeout
    public void openBrowser(String url, int timeout) {
        System.out.println("Opening browser with URL: " + url + " and 
timeout: " + timeout + " seconds");
    }
    // Overloaded method 3: Open browser with URL, timeout, and browser 
type
    public void openBrowser(String url, int timeout, String browserType) {
        System.out.println("Opening " + browserType + " browser with URL: " 
+ url + " and timeout: " + timeout + " seconds");
    }
}
```

Line-by-Line Breakdown:

1. **public void openBrowser(String url)**:
   This method takes a single url parameter and simulates opening a browser with the specified URL.

2. **public void openBrowser(String url, int timeout)**:
   This overloaded method accepts two parameters: url and timeout. It simulates setting a timeout for page load testing.

3. **public void openBrowser(String url, int timeout, String browserType)**:
   This version adds a browserType parameter, allowing users to specify the browser (e.g., Chrome, Firefox).

Execution Code:

```java
public class TestBrowserActions {
    public static void main(String[] args) {
        BrowserActions browserActions = new BrowserActions();

        // Using the first overloaded method
        browserActions.openBrowser("http://example.com");

        // Using the second overloaded method
        browserActions.openBrowser("http://example.com", 10);

        // Using the third overloaded method
        browserActions.openBrowser("http://example.com", 10, "Chrome");
    }
}
```

Execution Flow:

1. **Creating an instance of BrowserActions:**
   An object of the BrowserActions class is created to invoke its methods.

2. **Calling different overloaded methods:**
   The appropriate method is chosen based on the number of arguments passed, producing different outputs, such as:

    - Opening browser with URL: http://example.com
    - Opening browser with URL: http://example.com and timeout: 10 seconds
    - Opening Chrome browser with URL: http://example.com and timeout: 10 seconds

---

2. Run-Time Polymorphism (Method Overriding)

Definition:
Run-time polymorphism occurs when a subclass provides a specific implementation for a method that is defined in its superclass or interface. The method executed is determined at runtime.

Key Benefits in Selenium:

- Facilitates cross-browser testing.
- Promotes clean and reusable test logic.

Code Example 2: Cross-Browser Compatibility with WebDriver

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

public class BrowserTesting {
    public static void main(String[] args) {
        // WebDriver reference pointing to ChromeDriver instance
        WebDriver driver = new ChromeDriver();
        driver.get("http://example.com"); // ChromeDriver's implementation
of `get()`
        driver.quit();

        // WebDriver reference pointing to FirefoxDriver instance
        driver = new FirefoxDriver();
        driver.get("http://example.com"); // FirefoxDriver's implementation
of `get()`
        driver.quit();
    }
}
```

Line-by-Line Breakdown:

1. **WebDriver driver = new ChromeDriver();**
   A ChromeDriver object is created, but it is referenced as WebDriver, demonstrating polymorphism by treating ChromeDriver as a WebDriver.

2. **driver.get("http://example.com");**
   The get() method from ChromeDriver is called, loading the URL in the Chrome browser.

3. **driver.quit();**
   The quit() method is called to close the Chrome browser.

4. **Switching to FirefoxDriver:**
   A new FirefoxDriver object is assigned to the driver reference, and the same get() method is executed but this time in Firefox.

5. **Closing Firefox Browser:**
   The quit() method is called again to close the Firefox browser.

---

3. Real-World Selenium Framework Example

Polymorphism is extensively used in Selenium test frameworks to abstract common functionalities while handling specific behaviors of different web pages.

Framework Code Example

```java
import org.openqa.selenium.WebDriver;

public class BasePage {
    protected WebDriver driver;

    // Constructor
    public BasePage(WebDriver driver) {
        this.driver = driver;
    }

    // Reusable method
    public void navigateTo(String url) {
        driver.get(url);
    }
}
```

```
public class HomePage extends BasePage {
    public HomePage(WebDriver driver) {
        super(driver);
    }

    // Page-specific behavior
    public void search(String productName) {
        System.out.println("Searching for product: " + productName);
    }
}
```

Line-by-Line Breakdown:

1. **protected WebDriver driver;**
   Declares a WebDriver reference accessible within the class and its subclasses.

2. **public BasePage(WebDriver driver)**
   Constructor accepting a WebDriver object to initialize the driver variable. This ensures all pages can use the same WebDriver instance.

3. **public void navigateTo(String url)**
   A reusable method that navigates to a given URL. This method can be inherited by other page classes.

4. **public class HomePage extends BasePage**
   HomePage extends BasePage, inheriting the common navigateTo() method while adding its specific methods, such as search().

5. **super(driver);**
   The super(driver) call in the constructor initializes the driver for HomePage.

6. **public void search(String productName)**
   A method specific to HomePage, simulating a search functionality.

Execution Code

```
public class ECommerceTest {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        // Polymorphism: BasePage reference pointing to HomePage
        BasePage homePage = new HomePage(driver);

        // Calling BasePage method
        homePage.navigateTo("http://ecommerce.com");

        // Calling HomePage-specific method
        ((HomePage) homePage).search("Laptop");

        driver.quit();
    }
}
```

Explanation:

1. **homePage.navigateTo()** demonstrates the inheritance and reuse of the BasePage method.
2. **Casting BasePage to HomePage** allows access to the specific search() method.
