Implementing JUnit Solutions

The solutions would require implementing the logic in the respective classes (`PasswordValidator`, `DateUtils`, etc.) and then creating corresponding test methods in the test classes. Each test method should follow the Arrange-Act-Assert pattern:

1. Arrange: Set up the necessary environment or data.
2. Act: Call the method being tested.
3. Assert: Check that the result matches the expected outcome.

Exercise 1: Basic Function Testing

Objective: Write tests for a function that multiplies two integers.

```
Method: fun multiply(a: Int, b: Int): Int
```

Answer

```
class Calculator {
    fun multiply(a: Int, b: Int): Int = a + b
}

class CalculatorTest {

    @Test
    fun `multiplication of two numbers`() {
        val calculator = Calculator()
        assertEquals(4, calculator.multiply(2, 2))
    }
}
```

Explanation: This test verifies that the `add` method correctly calculates the sum of two integers. The `assertEquals` assertion checks if the result matches the expected value.

Exercise 2: Testing String Manipulation

Objective: Test a function that reverses a string.

```
fun reverse(str: String): String
```

Answer

```
class StringUtils {
    fun reverse(str: String): String = str.reversed()
}

class StringUtilsTest {

    @Test
    fun `reverse string`() {
        assertEquals("cba", StringUtils().reverse("abc"))
    }
}
```

Exercise 3: Validating Email Addresses

Objective: Test a function that checks if an email address is valid.

```
Method: fun isValidEmail(email: String): Boolean
```

Answer

```
class EmailValidator {
    fun isValidEmail(email: String): Boolean {
        // Simple regex for email validation
        return
android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()
    }
}

class EmailValidatorTest {

    @Test
    fun `valid email`() {

assertTrue(EmailValidator().isValidEmail("test@example.com"))
    }

    @Test
    fun `invalid email`() {
        assertFalse(EmailValidator().isValidEmail("test@invalid"))
    }
}
```

Explanation: These tests verify that the `isValidEmail` method correctly identifies valid and invalid email addresses.

Exercise 4: User Authentication

Objective: Test a method that authenticates a user based on username and password.

```
Class: UserAuthenticator with fun authenticate(username: String,
password: String): Boolean
```

Answer

```
// Assuming a simple UserAuthenticator class
class UserAuthenticator {
    fun authenticate(username: String, password: String): Boolean
{
        return username == "validUser" && password ==
"validPassword"
    }
}

class UserAuthenticatorTest {

    @Test
    fun `successful authentication`() {
        assertTrue(UserAuthenticator().authenticate("validUser",
"validPassword"))
    }

    @Test
    fun `failed authentication`() {

assertFalse(UserAuthenticator().authenticate("invalidUser",
"invalidPassword"))
    }
}
```

Explanation: These tests check if the `authenticate` method correctly returns `true` for valid credentials and `false` for invalid ones.

Exercise 5: RecyclerView Item Count

Objective: In an Android app, write a test to check if a `RecyclerView` adapter displays the correct number of items.

```
Class: YourCustomAdapter with getItemCount()
```

Answer

```
// Assuming a basic RecyclerView Adapter
class MyAdapter(private val items: List<String>) :
RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    // Implementation of Adapter methods...
    override fun getItemCount(): Int = items.size
}

class MyAdapterTest {

    @Test
    fun `adapter item count is correct`() {
        val adapter = MyAdapter(listOf("Item 1", "Item 2", "Item
3"))
        assertEquals(3, adapter.itemCount)
    }
}
```

Explanation: This test ensures that the `getItemCount` method in the adapter correctly reflects the number of items in the list.

Exercise 6: Testing a Password Strength Checker

Objective: Write a test for a function that checks the strength of a password based on certain criteria (length, numbers, uppercase characters).

```
class PasswordValidator {
    fun isPasswordStrong(password: String): Boolean {
}}
```

Answer

```
class PasswordValidator {
    fun isPasswordStrong(password: String): Boolean {
        // Implement password strength logic
       return password.length >= 8 && password.any { it.isDigit() } &&
password.any { it.isUpperCase() }

    }
}

class PasswordValidatorTest {

    @Test
    fun `password is strong`() {

assertTrue(PasswordValidator().isPasswordStrong("StrongPassword1"))
    }

    @Test
    fun `password is weak`() {
        assertFalse(PasswordValidator().isPasswordStrong("weak"))
    }
}
```

Explanation: The method checks if the password meets the minimum length requirement and contains at least one digit and one uppercase letter.

Exercise 7: Testing a Leap Year Calculator

Objective: Test a function that determines if a year is a leap year.

```
class DateUtils {
    fun isLeapYear(year: Int): Boolean {
        // Leap year if divisible by 4 but not by 100, or divisible by
400
    }
}
```

Answer

```
class DateUtils {
    fun isLeapYear(year: Int): Boolean {
        // Leap year calculation logic
       return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)

    }
}

class DateUtilsTest {

    @Test
    fun `year is a leap year`() {
        assertTrue(DateUtils().isLeapYear(2020))
    }

    @Test
    fun `year is not a leap year`() {
        assertFalse(DateUtils().isLeapYear(2019))
    }
}
```

Explanation: Leap years are years divisible by 4 but not by 100, except years divisible by 400. The method reflects this rule.

Exercise 8: Testing a User Age Validator

Objective: To check that the age validator correctly identifies valid and invalid ages (e.g., valid: 18-65).

Base class method:

```
class UserValidator {
    fun isAgeValid(age: Int): Boolean {
        return age in 18..65
    }
}
```

Answer

```
class UserValidator {
    fun isAgeValid(age: Int): Boolean {
        // Age validation logic
       return age in 18..65

    }
}

class UserValidatorTest {

    @Test
    fun `age is valid`() {
        assertTrue(UserValidator().isAgeValid(25))
    }

    @Test
    fun `age is invalid`() {
        assertFalse(UserValidator().isAgeValid(17))
    }
}
```

Explanation: The method checks if the age falls within a specified valid range (18 to 65 years in this case).