```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;
import static org.junit.jupiter.api.Assertions.*;
import java.time.Duration;

public class BankAccountTest {
    private BankAccount account;

    @BeforeEach
    public void setup() {
        account = new BankAccount("12345", 1000.0);
    }

    @Test
    public void testInitialBalance() {
        assertEquals(1000.0, account.getBalance(),
0.001, "Initial balance should be 1000.0");
    }

    @Test
    public void testDeposit() {
```

```
    @Test
    public void testInitialBalance() {
        assertEquals(1000.0, account.getBalance(),
0.001, "Initial balance should be 1000.0");
    }

    @Test
    public void testDeposit() {
        account.deposit(200.0);
        assertEquals(1200.0, account.getBalance(),
0.001, "Balance should be 1200.0 after depositing
200.0");
    }

    @Test
    public void testWithdraw() {
        account.withdraw(300.0);
        assertEquals(700.0, account.getBalance(),
0.001, "Balance should be 700.0 after withdrawing
300.0");
    }
```

```
    @Test
    public void testWithdrawInsufficientFunds() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.withdraw(2000.0);
        });
        assertEquals("Insufficient funds",
exception.getMessage(), "Exception message
should be 'Insufficient funds'");
    }


    @Test
    public void testNegativeInitialBalance() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            new BankAccount("54321", -100.0);
        });
        assertEquals("Initial balance cannot be
negative", exception.getMessage(), "Exception
message should be 'Initial balance cannot be
negative'");
```



```
    @Test
    public void testNegativeInitialBalance() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            new BankAccount("54321", -100.0);
        });
        assertEquals("Initial balance cannot be
negative", exception.getMessage(), "Exception
message should be 'Initial balance cannot be
negative'");
    }


    @Test
    public void testDepositNegativeAmount() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.deposit(-50.0);
        });
        assertEquals("Deposit amount must be
positive", exception.getMessage(), "Exception
message should be 'Deposit amount must be
positive'");
```

```
    @Test
    public void testDepositNegativeAmount() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.deposit(-50.0);
        });
        assertEquals("Deposit amount must be
positive", exception.getMessage(), "Exception
message should be 'Deposit amount must be
positive'");
    }


    @Test
    public void testWithdrawNegativeAmount() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.withdraw(-50.0);
        });
        assertEquals("Withdrawal amount must be
positive", exception.getMessage(), "Exception
message should be 'Withdrawal amount must be
positive'");
```

```
    @Test
    public void testWithdrawNegativeAmount() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.withdraw(-50.0);
        });
        assertEquals("Withdrawal amount must be
positive", exception.getMessage(), "Exception
message should be 'Withdrawal amount must be
positive'");
    }


    @Test
    public void testTransfer() {
        BankAccount targetAccount = new
BankAccount("67890", 500.0);
        account.transfer(targetAccount, 200.0);
        assertEquals(800.0, account.getBalance(),
0.001, "Balance should be 800.0 after transferring
200.0");
        assertEquals(700.0,
targetAccount.getBalance(), 0.001, "Target account
```

```
    @Test
    public void testTransfer() {
        BankAccount targetAccount = new BankAccount("67890", 500.0);
        account.transfer(targetAccount, 200.0);
        assertEquals(800.0, account.getBalance(), 0.001, "Balance should be 800.0 after transferring 200.0");
        assertEquals(700.0, targetAccount.getBalance(), 0.001, "Target account balance should be 700.0 after receiving 200.0");
    }

    @Test
    public void testTransferToNullAccount() {
        IllegalArgumentException exception = assertThrows(IllegalArgumentException.class, () -> {
            account.transfer(null, 100.0);
        });
        assertEquals("Target account cannot be null", exception.getMessage(), "Exception message should be 'Target account cannot be null'");
```

```
    @Test
    public void testTransferToNullAccount() {
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.transfer(null, 100.0);
        });
        assertEquals("Target account cannot be null",
exception.getMessage(), "Exception message
should be 'Target account cannot be null'");
    }


    @Test
    public void testTransferInsufficientFunds() {
        BankAccount targetAccount = new
BankAccount("67890", 500.0);
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.transfer(targetAccount, 2000.0);
        });
        assertEquals("Insufficient funds",
exception.getMessage(), "Exception message
```

```
    @Test
    public void testTransferInsufficientFunds() {
        BankAccount targetAccount = new
BankAccount("67890", 500.0);
        IllegalArgumentException exception =
assertThrows(IllegalArgumentException.class, () ->
{
            account.transfer(targetAccount, 2000.0);
        });
        assertEquals("Insufficient funds",
exception.getMessage(), "Exception message
should be 'Insufficient funds'");
    }

    @Test
    public void testNonNullAccountNumber() {
        assertNotNull(account.getAccountNumber(),
"Account number should not be null");
    }
```

```
    @Test
    public void testBalanceAfterMultipleOperations()
{
        account.deposit(500.0);
        account.withdraw(200.0);
        account.deposit(300.0);
        assertEquals(1600.0, account.getBalance(),
0.001, "Balance should be 1600.0 after multiple
operations");
    }

    @Test
    public void testAccountOperationsWithTimeout()
{
        assertTimeout(Duration.ofMillis(100), () -> {
            account.deposit(500.0);
            account.withdraw(200.0);
            account.deposit(300.0);
        }, "Account operations should complete within
100 milliseconds");
    }
```

```
    @Test
    public void testSameInstance() {
        BankAccount anotherAccount = account;
        assertSame(account, anotherAccount, "Both
references should point to the same instance");
    }

    @Test
    public void testNotSameInstance() {
        BankAccount anotherAccount = new
BankAccount("54321", 200.0);
        assertNotSame(account, anotherAccount,
"Both references should not point to the same
instance");
    }

    @Test
    public void testTrueCondition() {
        assertTrue(account.getBalance() > 0, "Balance
should be greater than 0");
    }

    @Test
```

```
    assertNotSame(account, anotherAccount,
"Both references should not point to the same
instance");
  }

  @Test
  public void testTrueCondition() {
    assertTrue(account.getBalance() > 0, "Balance
should be greater than 0");
  }

  @Test
  public void testFalseCondition() {
    account.withdraw(1000.0);
    assertFalse(account.getBalance() > 0, "Balance
should not be greater than 0 after withdrawing all
funds");
  }
}
```