TECH ELEVATOR

Java

Unit Testing
Exception Handling

Module 1 - Week 7

Housekeeping

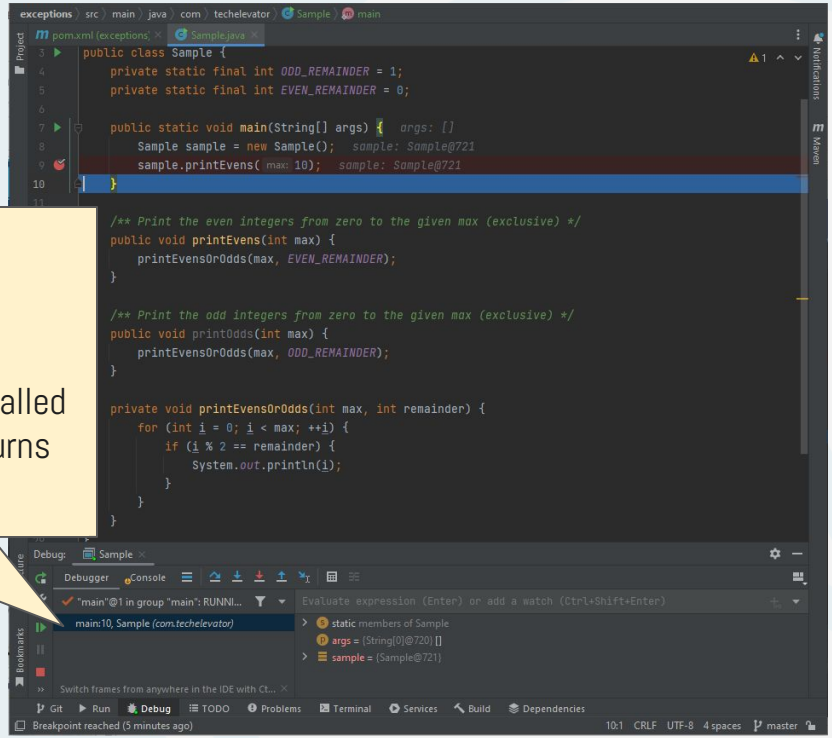
- If you received an email about gaining access to a new project this morning, you can ignore it.
- All Quizzes need to be completed before the end of the module.
- Technical interview questions and whiteboarding sessions.
- Absences reminder

Schedule

Week	Topics
Week 0	Welcome, Intro to Tools
Week 1	Variables and Data Types, Logical Branching
Week 2	Loops and Arrays, Command-Line Programs, Intro to Objects
Week 3	Collections
Week 4	**Mid-Module Project**
Week 5	Classes and Encapsulation
Week 6	Inheritance, Polymorphism
Week 7	Unit Testing, Exceptions and Error Handling
Week 8	File I/O Reading and Writing
Week 9	**End-of-module project**
Week 10	**Assessment**

Call Stack

- A **call stack** (aka execution stack) is a stack data structure that maintains information about the active methods of a computer program.
- When a method is called, it's pushed onto the stack. When it returns, it's popped from the stack.



This shows the call stack.

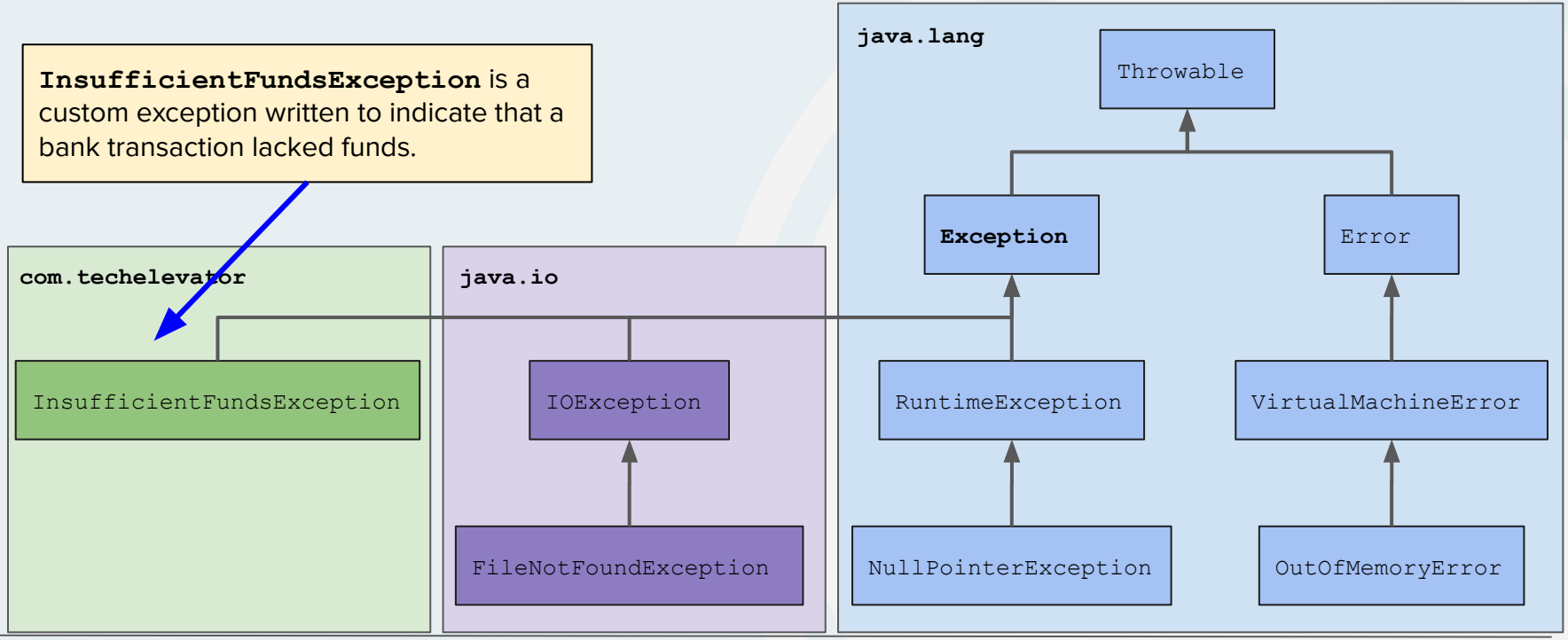
1. `main` is on the stack
2. `printEvens` is called
3. `printEvensOrOdds` is called
4. `printEvensOrOdds` returns
5. `printEvens` returns

Exceptions

- **Exceptions** are occurrences that alter the flow of the program away from the ideal or “happy” path.
- Sometimes it’s the developer’s fault, for example:
  - accessing an array element greater than the actual number of elements present
- Other times it’s not, for example:
  - loss of internet connection or a missing data file.
- Still other times, the developer uses exceptions to communicate anticipated possible errors that might occur in their code, for example:
  - not having enough money in an account to make a withdrawal

Exceptions

- Java uses classes inherited from the `java.lang.Exception` class to handle exception situations in code.
- We can write our own custom `Exception` classes by extending existing `Exception` classes.

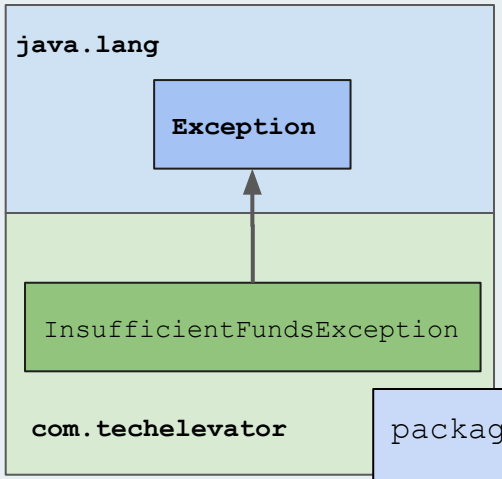


All `Exceptions` and `Errors` in Java are subclasses of the class `java.lang.Throwable`.

Exception	Error
An unexpected situations that occur while a program is executing. It is what happens when something is unexpected or goes wrong, such as the index of an array being out of bounds.	Used by the JVM to indicate errors that are associated with the runtime environment, such as running out of memory or other resources.
Possible to recover	Impossible to recover
Can be caught and handled	Should not be handled
Compile or runtime	Runtime
Caused by code or data	Caused by the running environment

Custom Exceptions

- Java enables us to write custom exceptions.
- InsufficientFundsException is an example of a custom exception written to indicate that a bank transaction lacked funds.



- `InsufficientFundsException` extends `java.lang.Exception`.
- Defined in the `com.techelevator` package.
- Defines a constructor that takes an integer indicating the money amount that triggered the exception.
- The constructor calls it's super constructor that takes a message argument to provide more meaningful information back to the user.

```
package com.techelevator;

public class InsufficientFundsException extends Exception {
    public InsufficientFundsException(int amount) {
        super("There is less than $" + amount + " in the account.");
    }
}
```

Try Catch

- In Java, we can write code to handle an exception by enclosing the code that might throw the exception in a `try` block.
- If an exception occurs within the `try` block, that exception can be handled by another block of code associated with the `try` block (the `catch` block).
- To associate the code that handles the exception with a `try` block, we put a `catch` block after the `try` block.
- Each `catch` block is an exception handler that handles the given type of exception.

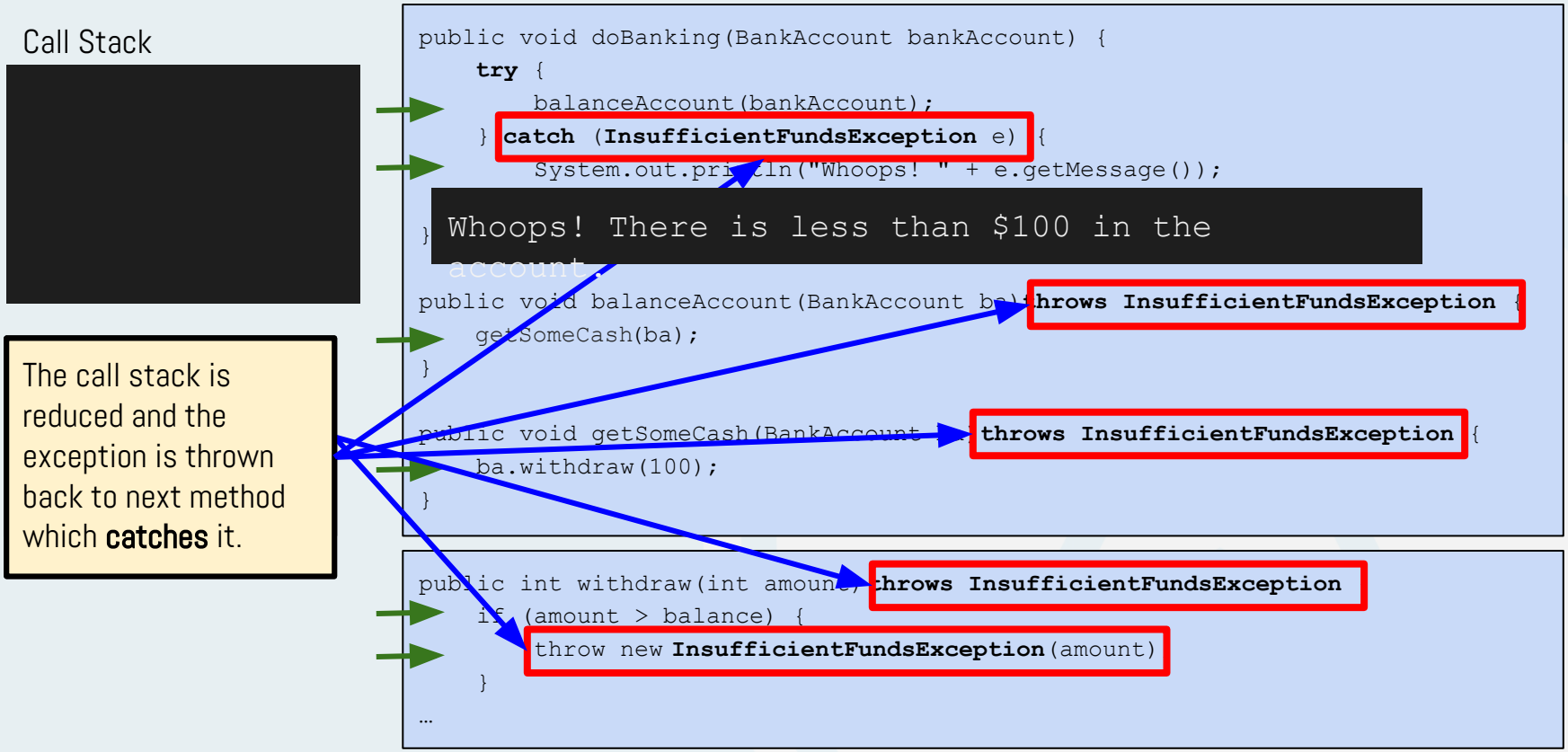
If the code in the **try** block throws an `InsufficientFundsException`...

... it will be handled here in the **catch** block.

```
public void doBanking() {
    BankAccount bankAccount = new BankAccount();
    try {
        balanceAccount(bankAccount);
        BigDecimal balance = bankAccount.getBalance();
        …
    } catch (InsufficientFundsException e) {
        System.out.println("Whoops! " + e.getMessage());
    }
}
```

Try Catch

A thrown exception "unwinds" the **call stack** until it is caught.



Try Catch Finally

- Multiple `catch` blocks can be associated with a `try` block to handle different types of exceptions.
- Once an exception is handled by a specific `catch` block, none of the other catch blocks will execute.
- A single `catch` block can handle more than one type of exception
- A `finally` block is ALWAYS executed after a `try` block. It allows us to avoid having cleanup code accidentally bypassed by a return or thrown exception
- Note that `finally` is not as useful since Java introduced the `try-with-resources` statement (more on this later).

Code in **`try`** block will attempt to execute.

If an exception of one of the given types is thrown from the **`try`** block, it will be handled in the appropriate **`catch`** block.

When the code in a **`try`** block is attempted, the finally block is **ALWAYS** executed whether or not an exception is thrown (even if the exception is handled in one of the **`catch`** blocks).

```
Resource resource = openResource();
try {
    // use the resource here...
} catch ( IndexOutOfBoundsException e) {
    // handle the IndexOutOfBoundsException here...
} catch (SQLException | IOException e) {
    // handle the SQLException or IOException here...
} finally {
    closeResource(resource);
}
```

Note that you can handle multiple exceptions in a single **`catch`** block.

Exception Polymorphism

- Exceptions are classes and can have member variables, methods, and multiple constructors like any other class.
- All exceptions (including `RuntimeException`) are inherited from the `java.lang.Exception`class.
- Polymorphism applies to exceptions (i.e `InvalidAmountException`"is a" `BankingException`).

```
public class BankingException extends Exception {
}
```

```
public class InvalidAmountException extends BankingException {
}
```

```
public class InsufficientFundsException extends BankingException {
}
```

Any subclass of **`BankingException`** can be treated as a **`BankingException`**.

This catch block will handle any thrown **`BankingException`** or subclass of a **`BankingException`**.

```
try {
    // code here...
} catch ( BankingException e) {
    // handle the BankingException here...
}
```

Checked Exceptions

- Java verifies checked exceptions at compile-time.
- If any code within a method throws a checked exception, then the method must either handle the exception or specify the exception with the throws keyword.

The **withdraw** method declares that it potentially throws a checked exception.

```
public int withdraw(int amount) throws InsufficientFundsException {
    if (amount > balance) {
        throw new InsufficientFundsException(amount);
    }
}
```

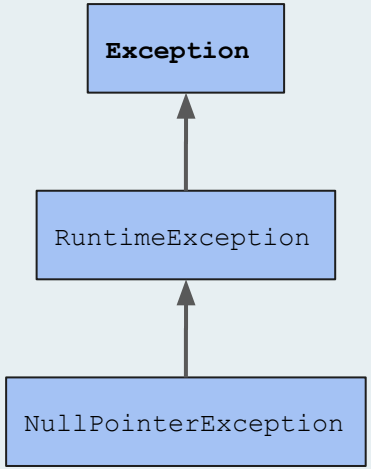
A method calling the **withdraw** method must handle it, or ...

```
public void doBanking(BankAccount bankAccount) {
    try {
        bankAccount.withdraw(100);
    } catch (InsufficientFundsException e) {
        System.out.println("Whoops! " + e.getMessage());
    }
}
```

... rethrow it

```
public void doBanking(BankAccount bankAccount)throws InsufficientFundsException {
    bankAccount.withdraw(100);
}
```

Unchecked Exceptions



- The `RuntimeException` class is the superclass of all unchecked exceptions.
- Unchecked exceptions are not checked at compile time.
- Unlike checked exceptions, you are not required to handle or specify the exception with the `throws` keyword.

If **`null`** is passed when this method is called, it will result in a **`NullPointerException`**.

`NullPointerException` is an unchecked exception so there is no requirement to handle it or specify it with the throws keyword.

```
public void doBanking(BankAccount bankAccount)throws InsufficientFundsException {
    bankAccount.withdraw(100);
}
```

Checked & Unchecked

Checked are compile-time exceptions
If code in a method throws a checked exception, the method must handle it or pass it up to the parent via a “throws” statement

Unchecked are run-time exceptions
User or code does something that causes the program to stop running
These are not checked at compile time; handle at the programmer’s discretion

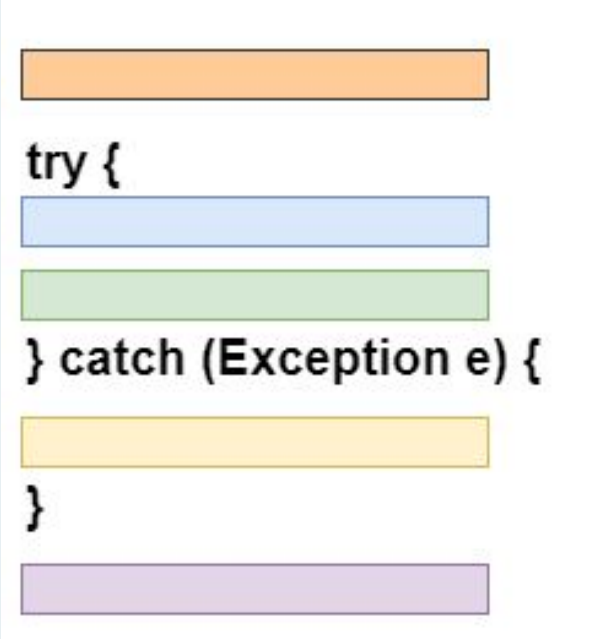
From Java documentation: “If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.”

Explain the parts and uses of the try-catch-finally block - Objective 2

Some common difficulties with exception handling are:

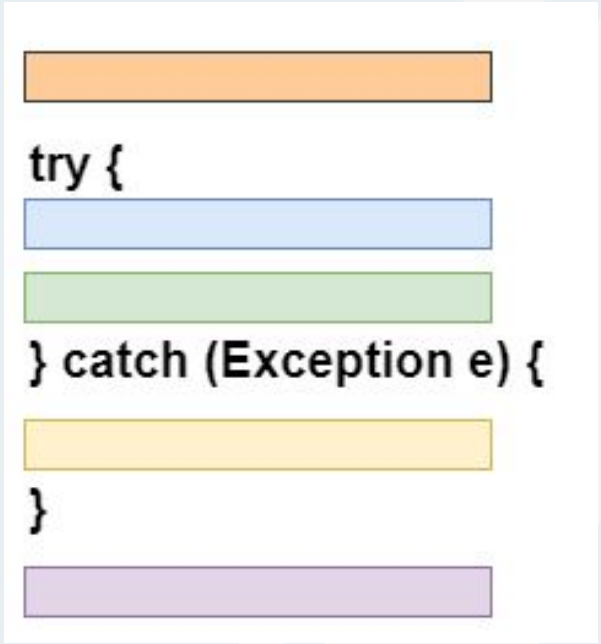
- wrapping too much code in a single try block
- mistakenly catching exceptions caused by errors within the program (bugs)
- putting catch blocks in the wrong order (a more general exception prior to a more specific one, for example)

Which blocks run when **no exceptions occur?**



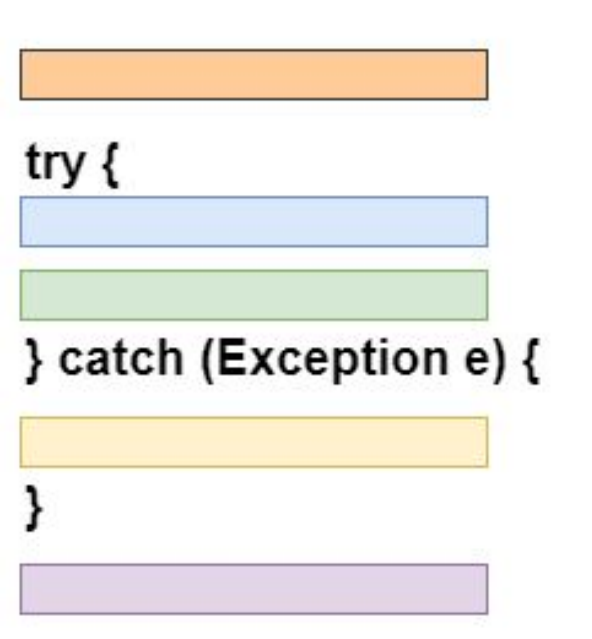
- Orange
- Blue
- Green
- Purple

Which blocks run when **an exception occurs in the blue box?**



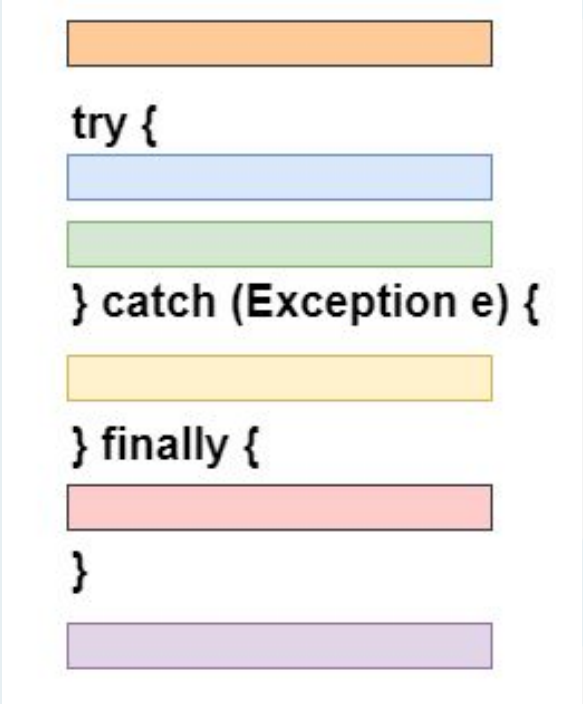
- Orange
- Blue
- Yellow
- Purple

Which blocks run when **an exception occurs in the blue block and the yellow block contains a return statement?**



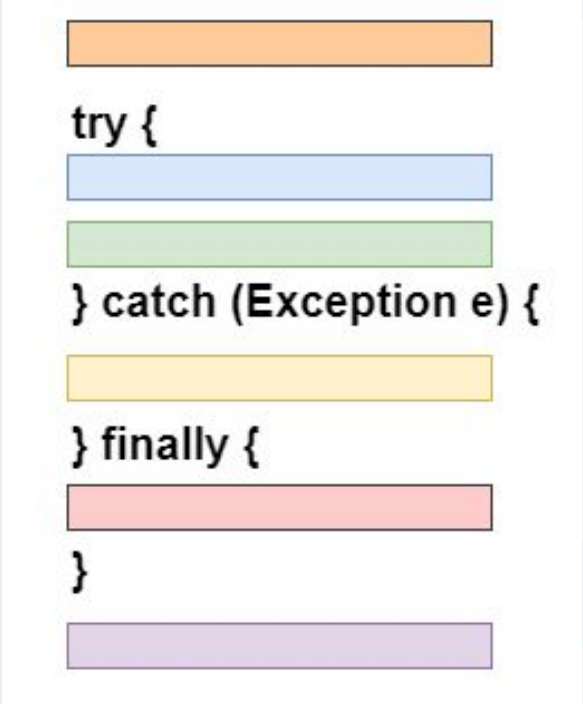
- Orange
- Blue
- Yellow

Which blocks run when **no exceptions occur?**



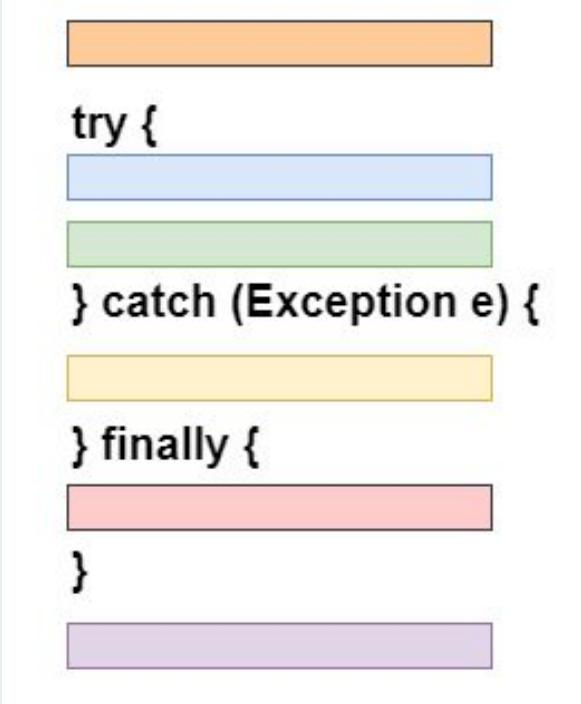
- Orange
- Blue
- Green
- Red
- Purple

Which blocks run when **an exception occurs in the blue box?**



- Orange
- Blue
- Yellow
- Red
- Purple

Which blocks run when **an exception occurs in the blue block and the yellow block contains a return statement?**



- Orange
- Blue
- Yellow
- Red

Break out rooms

- Join your team's breakout room and complete steps 1-5

Unit Testing

- **Unit Testing**: Tests the smallest units possible (i.e. methods of a class).
- The most commonly used testing framework in Java is **JUnit**.
- All related tests can be written in a single test class containing several test methods.
- Each method should contain at least one **assertion**, which compares the result of your code against an expected value.
- Why do unit testing?
  - Early Detection of Bugs and Defects
  - Improved Code Quality
  - Facilitates Code Refactoring
  - Enhanced Collaboration
  - Continuous Integration/Continuous Deployment (CI/CD) Support
  - Reduces Regression Issues
  - Increased Developer Confidence
  - Better Understanding of Requirements

Assertions

- JUnit leverages the concept of **assertions**.
- An **assertion** tests a condition and continues silently if the condition passes but fails the test with info about the condition that fails if the condition does not pass.

```
public class Example {
    public boolean isEven(int num) {
        return num % 2 == 0;
    }
}
```

```
@Test
public void isEven_withEvenValue_shouldReturnTrue() {
    Example example = new Example();
    boolean result = example.isEven(6);
    Assert.assertTrue(result);
}
```

Create object.

Call method with even number and store result.

Compares the expected to the actual result and fails the test if the assertion fails.

```
import org.junit.Assert;
…
@Test
public void isEven_expectTrue() {
    …
    Assert.assertTrue(result);
}
```

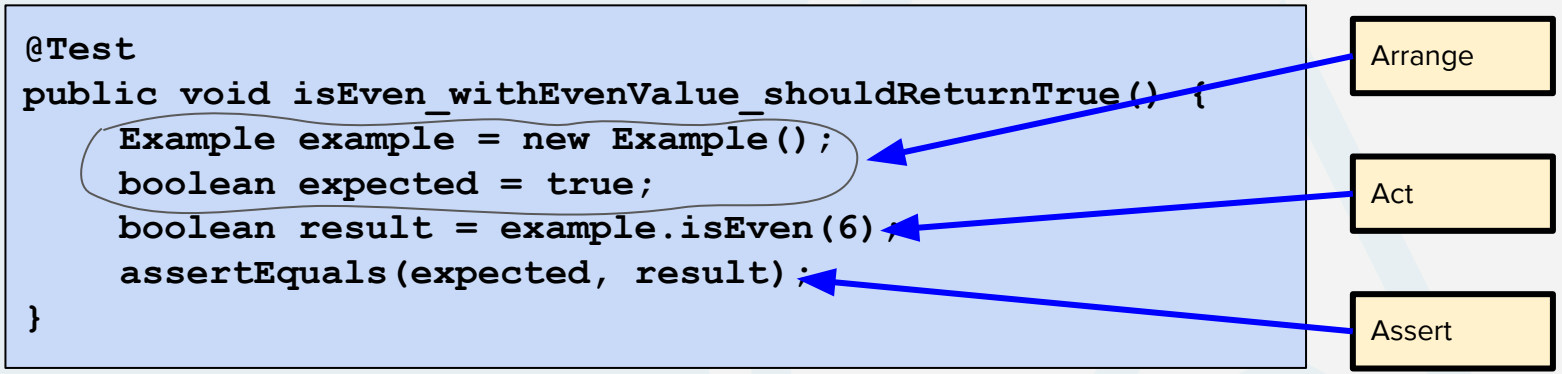
- assertEquals / assertNotEquals
- assertNotNull / assertNull
- assertTrue / assertFalse
- assertArrayEquals
- fail
- assertThat

More info on using asserts:

- https://www.baeldung.com/junit-assertions (JUnit4 portion applies to what we will be doing)
- https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

Unit Testing Structure

- **Arrange**: begin by arranging the conditions of the test, such as setting up test data
- **Act**: perform the action of interest, i.e. the thing we're testing
- **Assert**: validate that the expected outcome occurred by means of an assertion (e.g. a certain value was returned, a file exists, etc).



Test Cases

`@Test` is an annotation indicating this is a test.

There are two methods: one to test even case and one to test odd case. Tests are usually declared with `void` return type.

Evaluates result and fails if the assertion doesn't pass.

```
public class ExampleTest {
    @Test
    public void isEven_withEvenValue_shouldReturnTrue() {
        Example example = new Example();
        boolean result = example.isEven(6);
        Assert.assertTrue(result);
    }

    @Test
    public void isEven_withOddValue_shouldReturnFalse() {
        Example example = new Example();
        boolean result = example.isEven(9);
        Assert.assertFalse(result);
    }
}
```

Before and After

A method annotated with **`@Before`** will run before **each** test.

A method annotated with **`@After`** will run after **each** test.

Test flow:

1. **setup()**
2. first test
3. **tearDown()**
4. **setup()**
5. second test
6. **tearDown()**

```
public class ExampleTest {

      private Example example;

      @Before
      public void setUp() {
            example = new Example();  // do test setup
      }

      @After
      public void tearDown() {
            example = null;  // do test cleanup
      }

      @Test
      public void isEven_withEvenValue_shouldReturnTrue() {
            Assert.assertTrue(example.isEven(6));
      }

      @Test
      public void isEven_withOddValue_shouldReturnFalse() {
            Assert.assertFalse(example.isEven(9));
      }
}
```

Unit Test Best Practices

- No external dependencies
- Each test should only contain one concept.
- **arrange** / **act** / **assert**
- Test code should be of the same quality as production code

Some common difficulties with unit tests are:

- identifying edge cases to test
- structuring code in a way that makes it testable
- knowing which test code is part of the "arrange" step vs. the "act" step
- understanding the delta that's required when using assertEquals() with double values
- assertions involving arrays and objects rather than primitive values

Break out rooms

- Join your team's breakout room and complete step 6