# Introduction to Testing

# Testing

- testing code is a vital part of the development process
- the goal of testing is to find defects in your code
  - Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
    —Edsger W. Dijkstra

- how can we test our utility class?
  - write a program that uses it and verify the result

```
public class IsThreeOfAKindTest {
  public static void main(String[] args) {
    // make a list of 5 dice that are 3 of a kind
    // check if Yahtzee.isThreeOfAKind returns true
  }
}
```

```
public class IsThreeOfAKindTest {
  public static void main(String[] args) {
    // make a list of 5 dice that are 3 of a kind
    List<Die> dice = new ArrayList<Die>();
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 2));   // 2
    dice.add(new Die(6, 3));   // 3

    // check if Yahtzee.isThreeOfAKind returns true
  }
}
```

```
public class IsThreeOfAKindTest {
  public static void main(String[] args) {
    // make a list of 5 dice that are 3 of a kind
    List<Die> dice = new ArrayList<Die>();
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 2));   // 2
    dice.add(new Die(6, 3));   // 3

    // check if Yahtzee.isThreeOfAKind returns true
    if (Yahtzee.isThreeOfAKind(dice) == true) {
      System.out.println("success");
    }
  }
}
```

```
public class IsThreeOfAKindTest {
  public static void main(String[] args) {
    // make a list of 5 dice that are 3 of a kind
    List<Die> dice = new ArrayList<Die>();
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 2));   // 2
    dice.add(new Die(6, 3));   // 3

    // check if Yahtzee.isThreeOfAKind returns false
    if (Yahtzee.isThreeOfAKind(dice) == false) {
      throw new RuntimeException("FAILED: " +
          dice + " is a 3-of-a-kind");
    }
  }
}
```

# Testing

- we tested one case with a specific list of dice and an expected return value
  - the dice 1, 1, 1, 2, 3
  - expect method to return **true**

- the method we are testing should return false if the list of dice does not contain a three-of-a-kind
  - we should test this, too

# Testing

- checking if a test fails and throwing an exception makes it easy to find tests that fail
  - because uncaught exceptions terminate the running program
  - unfortunately, stopping the test program might mean that other tests remain unrunnable
    - at least until you fix the broken test case

```
public class IsNotThreeOfAKindTest {
  public static void main(String[] args) {
    // make a list of 5 dice that are not 3 of a kind
    List<Die> dice = new ArrayList<Die>();
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 6));   // 6
    dice.add(new Die(6, 2));   // 2
    dice.add(new Die(6, 3));   // 3

    // check if Yahtzee.isThreeOfAKind returns true
    if (Yahtzee.isThreeOfAKind(dice) == true) {
      throw new RuntimeException("FAILED: " +
          dice + " is not a 3-of-a-kind");
    }
  }
}
```

# Unit Testing

- A unit test examines the behavior of a distinct unit of work. Within a Java application, the "distinct unit of work" is often (but not always) a single method. ... A unit of work is a task that isn't directly dependent on the completion of any other task."
  - from the book JUnit in Action

# Unit Testing: Test Cases

- a unit test consists of one or more *test cases*
- a test case for a method consists of
  - a specific set of arguments for the method
  - the expected output or result of the method

# Unit Testing: Test Cases

- example test case 1 for **`isThreeOfAKind`**
  - a specific set of arguments for the method
    - the list of dice with values 1, 1, 1, 2, 3
  - the expected output or result of the method
    - **true**

- example test case 2 for **`isThreeOfAKind`**
  - a specific set of arguments for the method
    - the list of dice with values 1, 1, 6, 2, 3
  - the expected output or result of the method
    - **false**

# Unit Testing: Test Cases

- a test case passes if the actual output of the method matches the expected output of the method when the method is run with the test case arguments
  - it fails if the actual output does not match the expected output

- typically, you use several test cases to test a method
  - the course notes uses the term *test vector* to refer to a collection of test cases
  - the term *test suite* is also commonly used

# JUnit

- JUnit is a testing framework for Java
- JUnit provides a way for creating:
  - test cases
    - a class that contains one or more tests
  - test suites
    - a group of test cases
  - test runner
    - a way to automatically run test suites

```java
import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.List;

import org.junit.Test;

public class YahtzeeTest {

  @Test
  public void isThreeOfAKind() {
    // make a list of 5 dice that are 3 of a kind
    List<Die> dice = new ArrayList<Die>();
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 2));   // 2
    dice.add(new Die(6, 3));   // 3

    assertTrue(Yahtzee.isThreeOfAKind(dice));
  }
```

# JUnit

- our unit test tests if **`isThreeOfAKind`** produces the correct answer (**`true`**) if the list contains a three of a kind
- we should also test if **`isThreeOfAKind`** produces the correct answer (**`false`**) if the list *does not* contain a three of a kind

```
  @Test
  public void notThreeOfAKind() {
    // make a list of 5 dice that are not 3 of a kind
    List<Die> dice = new ArrayList<Die>();
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 1));   // 1
    dice.add(new Die(6, 6));   // 6
    dice.add(new Die(6, 2));   // 2
    dice.add(new Die(6, 3));   // 3

    assertFalse(Yahtzee.isThreeOfAKind(dice));
  }

}
```

# JUnit

- our unit tests use specific cases of rolls:
  - 1, 1, 1, 2, 3 **`isThreeOfAKind`**
  - 1, 1, 6, 2, 3 **`notThreeOfAKind`**
- the tests don't tell us if our method works for different rolls:
  - 3, 2, 1, 1, 1 ?
  - 4, 6, 2, 3, 5 ?
- can you write a unit test that tests every possible roll that is a three of a kind? every possible roll that is not three of a kind?

# JUnit

- notice that our test tests one specific three-of-a-kind
  - 1, 1, 1, 2, 3
- shouldn't we test all possible three-of-a-kinds?
  - or at least more three-of-a-kinds

- how can you generate a list of dice that is guaranteed to contain three-of-a-kind?

# JUnit

- notice that our test tests one specific three-of-a-kind
  - 1, 1, 6, 2, 3
- shouldn't we test all possible not-three-of-a-kinds?
  - or at least more not-three-of-a-kinds

- how can you generate a list of dice that is guaranteed to *not* contain three-of-a-kind?

# Limitations of testing

- who tests the tests?
- it is often impossible or impractical to test all possible combinations of values for the method parameters
  - this means that the test implementer must choose specific test cases
    - it is very easy to miss test cases that would reveal an error
      - your code might contain an error even if your code passes all of the tests

# Another testing example

- consider the following utility class that checks if a numeric value lies inside a specified range

```java
public class InRange {

  /**
   * Returns true if value is in the range [lo, hi].
   * value is inside the range if and only if
   * lo <= value <= hi
   *
   * @param lo the lower endpoint of the range
   * @param hi the higher endpoint of the range
   * @param value the value to check
   * @pre. lo <= hi
   * @return true if lo <= value <= hi, and false otherwise
   */
  public static boolean isInRange(int lo, int hi, int value) {
    return value > lo && value < hi;
  }
}
```

# Test cases

- what test cases should we use to test our method?

# Readings

- the lectures have covered most of Chapter 2 of the course notes
  - up to and including all of Section 2.5
  - you should review the course notes to make sure you understand all of the material through Section 2.5
- try to read Section 2.6 *Beyond the Basics*
  - some of the material from pages 34 and on are quite challenging