Utilities (Part 3)

Implementing static features

Goals for Today

- learn about preconditions versus validation
- introduction to documentation
- introduction to testing

Yahtzee class so far

- recall our implementation of the Yahtzee class so far
  - private constructor to prevent instantiation
  - public constant field that represents the number of dice
  - public method that determines if a list of dice represents a roll of three-of-a-kind

```java
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;

public class Yahtzee {

  private Yahtzee() {
    // private and empty by design
  }

  public static final int NUMBER_OF_DICE = 5;

  public static boolean isThreeOfAKind(List<Die> dice) {
    List<Die> copy = new ArrayList<Die>(dice);
    Collections.sort(copy);
    boolean result = copy.get(0).getValue() == copy.get(2).getValue() ||
                     copy.get(1).getValue() == copy.get(3).getValue() ||
                     copy.get(2).getValue() == copy.get(4).getValue();
    return result;
  }

}
```

Yahtzee client: Not enough dice

- consider the following client program that tries to use our utility class using fewer than 5 dice

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

public class YahtzeeClient {
  public static void main(String[] args) {
    final int N_DICE = 3;                          // NOT ENOUGH DICE
    List<Die> dice = new ArrayList<Die>();
    for (int i = 0; i < N_DICE; i++) {
      dice.add(new Die());
    }
    System.out.print("Dice: " + dice.get(0).getValue());
    for (int i = 1; i < N_DICE; i++) {
      System.out.print(", " + dice.get(i).getValue());
    }
    System.out.println();
    boolean isThree = Yahtzee.isThreeOfAKind(dice);
    System.out.println("three of a kind?: " + isThree);
  }
}
```

Yahtzee client: Not enough dice

- the output of the program is:

```
Dice: 5, 4, 4
Exception in thread "main"
java.lang.IndexOutOfBoundsException: Index: 3, Size: 3
at java.util.ArrayList.RangeCheck(Unknown Source)
at java.util.ArrayList.get(Unknown Source)
at Yahtzee.isThreeOfAKind(Yahtzee.java:38)
at YahtzeeClient.main(YahtzeeClient.java:19)
```

Yahtzee client: Too many dice

- consider the following client program that tries to use our utility class using more than 5 dice

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

public class YahtzeeClient {
  public static void main(String[] args) {
    final int N_DICE = 7;                        // TOO MANY DICE
    List<Die> dice = new ArrayList<Die>();
    for (int i = 0; i < N_DICE; i++) {
      dice.add(new Die());
    }
    System.out.print("Dice: " + dice.get(0).getValue());
    for (int i = 1; i < N_DICE; i++) {
      System.out.print(", " + dice.get(i).getValue());
    }
    System.out.println();
    boolean isThree = Yahtzee.isThreeOfAKind(dice);
    System.out.println("three of a kind?: " + isThree);
  }
}
```

Yahtzee client: Too many dice

- the program seems to work sometimes:

```
Dice: 3, 2, 2, 5, 2, 4, 1
three of a kind?: true
```

- but fails sometimes:

```
Dice: 6, 3, 3, 6, 6, 5, 5
three of a kind?: false
```

Preconditions and postconditions

- recall the meaning of method pre- and postconditions
- precondition
  - a condition that the client must ensure is true immediately before a method is invoked
- postcondtion
  - a condition that the method must ensure is true immediately after the method is invoked

Who is responsible?

- our method **`isThreeOfAKind`** clearly fails if the client uses the wrong number of dice
    - we say that the method cannot satisfy its *postcondition* if the client uses the wrong number of dice
- as the implementer, we should advertise this fact as part of the method API
- as the implementer, we also need to decide who is responsible if a client uses the wrong number of dice

Client is responsible: Preconditions

- as the implementer, we can choose to make the client responsible for errors caused by using the wrong number of dice
- we do this by stating in the API that the method has a *precondition*
  - we'll see exactly how to do this in Java shortly

Client is responsible: Preconditions

- recall that a method precondition is a condition that the client must ensure is true immediately before invoking a method
  - if the precondition is not true, then the client has no guarantees of what the method will do
- for utility class methods, preconditions are conditions on the values of the arguments passed to the method
  - e.g., in our current implementation of **`isThreeOfAKind`** the number of dice must be 5

Implementer is responsible: Validation

- as the implementer, we can choose to specify precisely what happens if the method cannot satisfy its postcondition given the arguments provided by the client
- this often requires that the method implementation validate its parameters
  - e.g., **`isThreeOfAKind`** would have to check that the client has used a list argument containing 5 dice

```java
public static boolean isThreeOfAKind(List<Die> dice) {
  if (dice.size() != Yahtzee.NUMBER_OF_DICE) {
      throw new IllegalArgumentException("wrong number of dice: " +
                                         dice.size());
  }
  List<Die> copy = new ArrayList<Die>(dice);
  Collections.sort(copy);
  boolean result = copy.get(0).getValue() == copy.get(2).getValue() ||
                   copy.get(1).getValue() == copy.get(3).getValue() ||
                   copy.get(2).getValue() == copy.get(4).getValue();
  return result;
}
```

Documenting

- documenting code was not a new idea when Java was invented
  - however, Java was the first major language to embed documentation in the code and extract the documentation into readable electronic APIs

- the tool that generates API documents from comments embedded in the code is called Javadoc

Documenting

- Javadoc processes *doc comments* that immediately precede a class, attribute, constructor or method declaration
  - doc comments delimited by `/**` and `*/`
  - doc comment written in HTML and made up of two parts
    1. a description
       - first sentence of description gets copied to the summary section
       - only one description block; can use `<p>` to create separate paragraphs
    2. block tags
       - begin with `@` (`@param`, `@return`, `@throws` and many others)
       - `@pre.` is a non-standard (custom tag used in EECS1030) for documenting preconditions

Method documentation example

Eclipse will generate an empty Javadoc comment for you if you right-click on the method header and choose **Source→Generate Element Comment**

```
/**
 *
 * @param dice
 * @return
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Method documentation example

The first sentence of the documentation should be short summary of the method; this sentence appears in the method summary section.

```
/**
 * Returns true if the list dice contains a three-of-a-kind.
 *
 * @param dice
 * @return
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Method documentation example

If you want separate paragraphs in your documentation, you need to use the html paragraph tag **<p>** to start a new paragraph.

```
/**
 * Returns true if the list dice contains a three-of-a-kind.
 *
 * <p>A three of a kind is defined as at least three dice having
 * the same value.
 *
 * @param dice
 * @return
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Method documentation example

You should provide a brief description of each parameter.

```
/**
 * Returns true if the list dice contains a three-of-a-kind.
 *
 * <p>A three of a kind is defined as at least three dice having
 * the same value.
 *
 * @param dice list of dice representing the roll
 * @return
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Method documentation example

Provide a brief description of the return value if the return type is not void. This description often describes a postcondition of the method.

```java
/**
 * Returns true if the list dice contains a three-of-a-kind.
 *
 * <p>A three of a kind is defined as at least three dice having
 * the same value.
 *
 * @param dice list of dice representing the roll
 * @return true if dice contains three-of-a-kind, false otherwise
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Method documentation example

- if a method has one or more preconditions, you should use the EECS1011 specific **@pre.** tag to document them
  - e.g., if we were documenting our original version of isThreeOfAKind we would use an **@pre.** tag to document the precondition
    `dice.size() == Yahtzee.NUMBER_OF_DICE`

Method documentation example

Describe any preconditions using the EECS1011 specific **@pre.** tag.

```
/**
 * Returns true if the list dice contains a three-of-a-kind.
 *
 * <p>A three of a kind is defined as at least three dice having
 * the same value.
 *
 * @param dice list of dice representing the roll
 * @pre. dice.size() == Yahtzee.NUMBER_OF_DICE
 * @return true if dice contains three-of-a-kind, false otherwise
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Method documentation example

- if a method throws an exception (perhaps as a result of failing to validate a parameter) then you should use the **@throws** tag to document the exception
  - e.g., if we were documenting our second version of **`isThreeOfAKind`** we would use the **@throws** tag to document the exception that is thrown if **`dice.size() != Yahtzee.NUMBER_OF_DICE`**

Method documentation example

Use a **@throws** tag to document each exception that might be thrown by your method.

```
/**
 * Returns true if the list dice contains a three-of-a-kind.
 *
 * <p>A three of a kind is defined as at least three dice having
 * the same value.
 *
 * @param dice list of dice representing the roll
 * @return true if dice contains three-of-a-kind, false otherwise
 * @throws IllegalArgumentException if dice.size() !=
           Yahtzee.NUMBER_OF_DICE
 */
public static boolean isThreeOfAKind(List<Die> dice) {
  // implementation not shown
}
```

Documenting fields

- all public fields should have a Javadoc comment describing the field
  - Eclipse will generate an empty Javadoc comment for you if you right-click on the field declaration and choose **Source→Generate Element Comment**

Field documentation example

```
public class Yahtzee {

  /**
   * The number of six-sided dice used in a standard game
   * of Yahtzee.
   */
  public static final int NUMBER_OF_DICE = 5;
```

Documenting classes

- all classes should contain a description of the class
  - Eclipse will generate an empty Javadoc comment for you if you right-click on the field declaration and choose **Source→Generate Element Comment**
- the description of a class can be quite detailed for sophisticated classes
  - e.g., java.lang.String
- you should describe the purpose of the class and any other information that might be important to clients
  - but normally you do not describe the implementation details of the class

Class documentation example

```
/**
 * A utility class that encodes a subset of the rules for
 * the game Yahtzee.
 *
 * <p>A description of the scoring categories can be
 * found on the <a href="http://en.wikipedia.org/wiki/Yahtzee">
 * Yahtzee Wikipedia web page</a>.
 *
 * @author EECS1011E_W15
 *
 */
public class Yahtzee {
   // implementation not shown
}
```

javadoc Documentation

- Oracle's how-to page
  - http://www.oracle.com/technetwork/articles/java/index-137868.html
- also see the examples in the course notes