

Utilities (Part 2)

Implementing static features

Goals for Today

- ▶ learn about preventing class instantiation
- ▶ learn what a utility is in Java
- ▶ learn about implementing methods
 - ▶ static methods
 - ▶ pass-by-value
- ▶ Javadoc

Puzzle 2

- ▶ what does the following program print?

```
public class Puzzle02
{
    public static void main(String[] args)
    {
        final long
            MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
        final long
            MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
        System.out.println(MICROS_PER_DAY / MILLIS_PER_DAY);
    }
}
```

- ▶ prints 5
- ▶ the problem occurs because the expression

`24 * 60 * 60 * 1000 * 1000`

evaluates to a number bigger than `int` can hold

- ▶ `86,400,000,000 > 2,147,483,647 (Integer.MAX_VALUE)`
- ▶ called *overflow*
- ▶ notice that the numbers in the expression are of type `int`
 - ▶ Java will evaluate the expression using `int` even though the constant `MICROS_PER_DAY` is of type `long`
- ▶ solution: make sure that the first value matches the destination type

`24L * 60 * 60 * 1000 * 1000`

Overflow

- ▶ several well known problems caused by issues related to overflow
 - ▶ Year 2000 problem
 - ▶ Year 2038 problem
 - ▶ Ariane 5 Flight 501

new Yahtzee Objects

- ▶ our **Yahtzee** API does not expose a constructor
 - ▶ but

```
Yahtzee y = new Yahtzee();
```

is legal

- ▶ if you do not define any constructors, Java will generate a default no-argument constructor for you
 - ▶ e.g., we get the **public** constructor

```
public Yahtzee() { }
```

even though we did not implement it

Preventing Instantiation

- ▶ our **Yahtzee** API exposes only **static** constants (and methods later on)
 - ▶ its state is constant
- ▶ there is no benefit in instantiating a **Yahtzee** object
 - ▶ a client can access the constants (and methods) without creating a **Yahtzee** object

```
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

- ▶ can prevent instantiation by declaring a **private** constructor

Version 2 (prevent instantiation)

```
public class Yahtzee {  
    // fields  
    public static final int NUMBER_OF_DICE = 5;  
  
    // constructors  
    // suppress default ctor for non-instantiation  
    private Yahtzee() {  
    }  
  
}
```

[notes 1.2.3]



Version 2.1 (even better)

```
public class Yahtzee {
    // fields
    public static final int NUMBER_OF_DICE = 5;

    // constructors
    // suppress default ctor for non-instantiation
    private Yahtzee() {
        throw new AssertionError();
    }
}
```

[notes 1.2.3]

private

- ▶ **private** fields, constructors, and methods cannot be accessed by clients
 - ▶ they are not part of the class API
- ▶ **private** fields, constructors, and methods are accessible only inside the scope of the class
- ▶ a class with only **private** constructors indicates to clients that they cannot use **new** to create instances of the class

Utilities

- ▶ in Java, a *utility* class is a class having only static fields and static methods
- ▶ uses:
 - ▶ group related methods on primitive values or arrays
 - ▶ `java.lang.Math` or `java.util.Arrays`
 - ▶ group static methods for objects that implement an interface
 - ▶ `java.util.Collections`
 - ▶ [notes 1.6.1–1.6.3]
 - ▶ group static methods on a **final** class
 - ▶ more on this when we talk about inheritance

```
public class Yahtzee {
    // fields
    public static final int NUMBER_OF_DICE = 5;

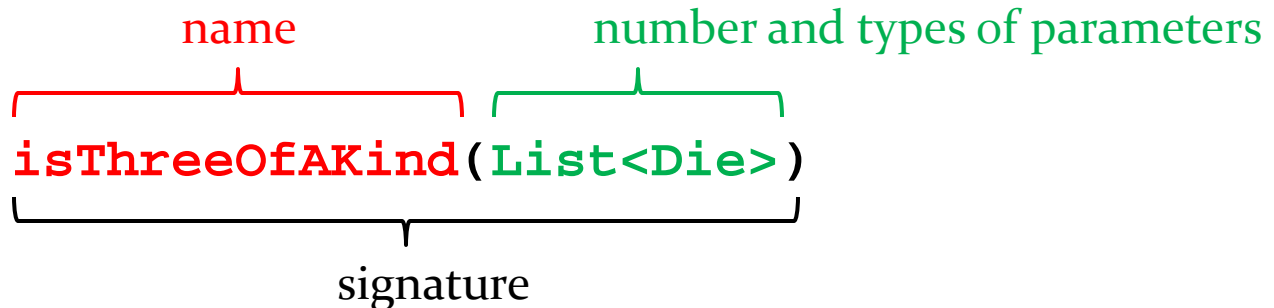
    // constructors
    // suppress default ctor for non-instantiation
    private Yahtzee() {
        throw new AssertionError();
    }

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

Method Signatures

```
public static boolean isThreeOfAKind(List<Die> dice)
```

- ▶ a method is a member that performs an action
- ▶ a method has a signature (name + number and types of the parameters)



- ▶ all method signatures in a class must be unique

Method Signatures

- ▶ what happens if we try to introduce a second method

```
public static boolean
```

```
    isThreeOfAKind(Collection<Integer> dice) ?
```

- ▶ what about

```
public static boolean
```

```
    isThreeOfAKind(List<Integer> dice) ?
```

Methods

```
public static boolean isThreeOfAKind(List<Die> dice)
```

- ▶ a method returns a typed value or `void`

`boolean`

- ▶ use `return` to indicate the value to be returned

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

Parameters

- ▶ sometimes called *formal parameters*
- ▶ for a method, the parameter names must be unique
 - ▶ but a parameter can have the same name as an attribute (see [notes 1.3.3])
- ▶ the scope of a parameter is the body of the method

static Methods

- ▶ a method that is **static** is a per-class member
 - ▶ client does not need an object to invoke the method
 - ▶ client uses the class name to access the method

```
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

- ▶ **static** methods are also called *class methods*
- ▶ a **static** method can only use **static** fields of the class

[notes 1.2.4], [A] 249-255]

Invoking Methods

- ▶ a client invokes a method by passing arguments to the method
- ▶ the types of the arguments must be compatible with the types of parameters in the method signature
- ▶ the values of the arguments must satisfy the preconditions of the method contract [JBA 2.3.3]

```
List<Die> dice = new ArrayList<Die>();  
for (int i = 0; i < 5; i++) {  
    dice.add(new Die());  
} argument  
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

Pass-by-value

- ▶ Java uses pass-by-value to:
 - ▶ transfer the value of the arguments to the method
 - ▶ transfer the return value back to the client
- ▶ consider the following utility class and its client...

```
import type.lib.Fraction;

public class Doubler {

    private Doubler() {
    }

    // tries to double x
    public static void twice(int x) {
        x = 2 * x;
    }

    // tries to double f
    public static void twice(Fraction f) {
        long numerator = f.getNumerator();
        f.setNumerator( 2 * numerator );
    }
}
```

```
import type.lib.Fraction;

public class TestDoubler {

    public static void main(String[] args) {
        int a = 1;
        Doubler.twice(a);

        Fraction b = new Fraction(1, 2);
        Doubler.twice(b);

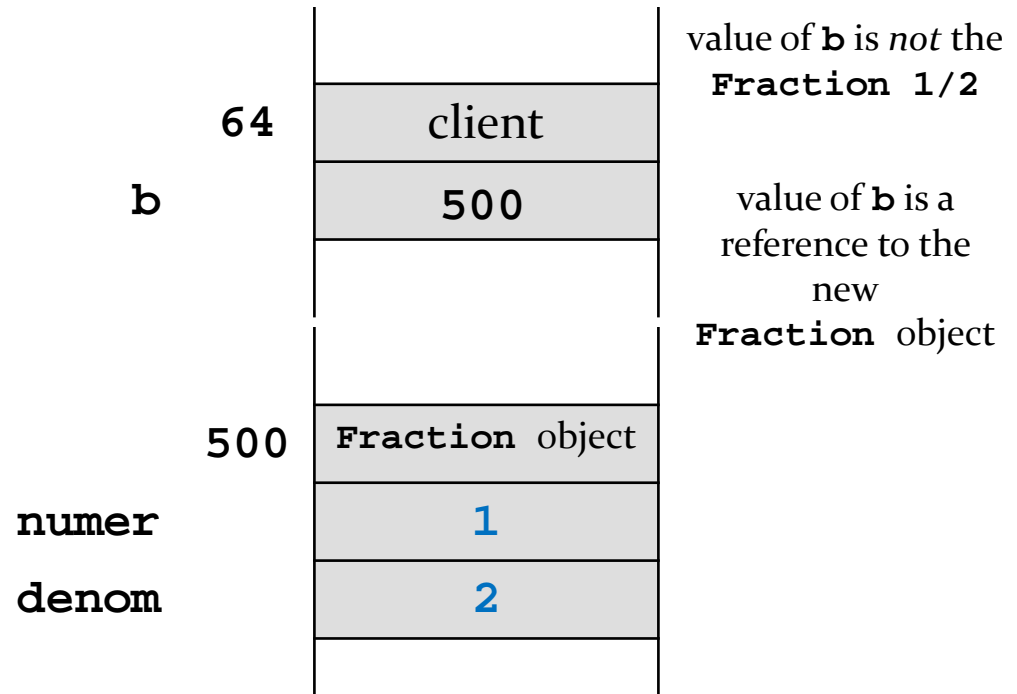
        System.out.println(a);
        System.out.println(b);
    }
}
```

Pass-by-value

- ▶ what is the output of the client program?
 - ▶ try it and see
- ▶ an invoked method runs in its own area of memory that contains storage for its parameters
- ▶ each parameter is initialized with *the value* of its corresponding argument

Pass-by-value with Reference Types

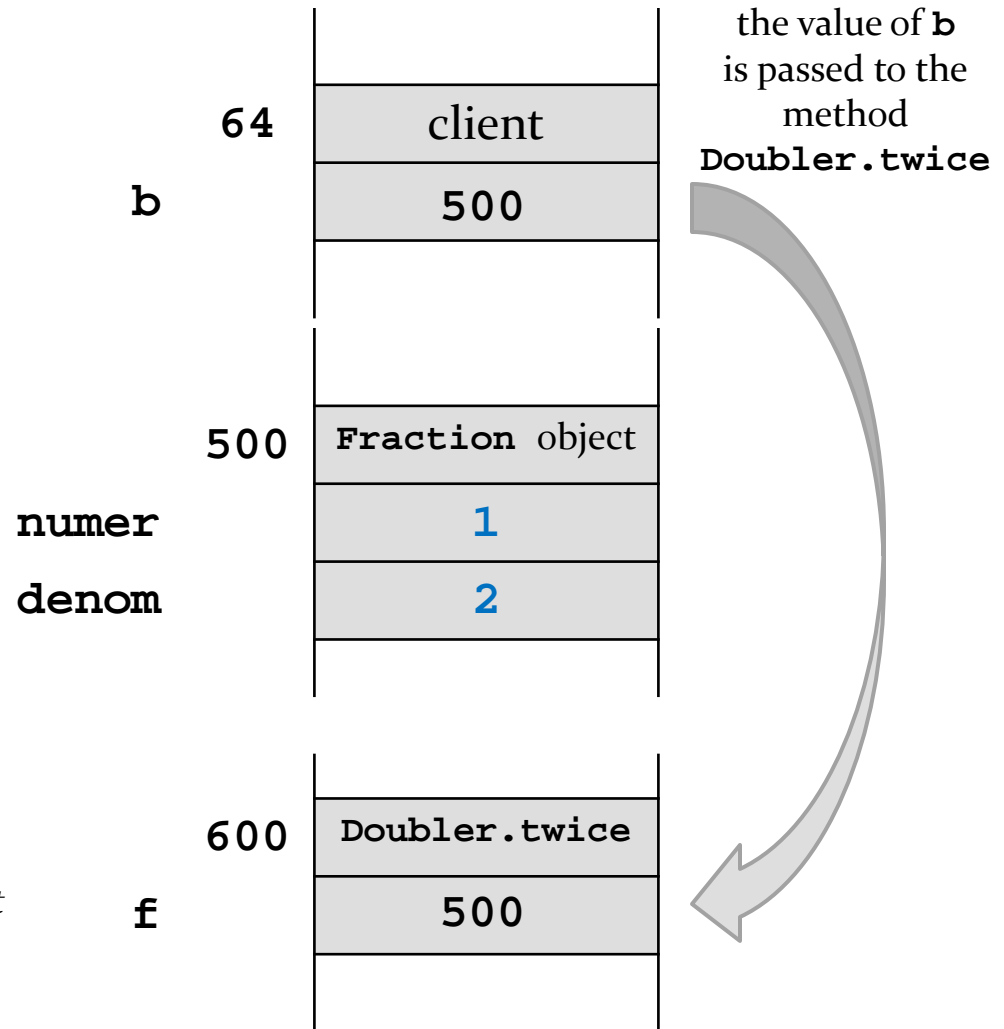
```
Fraction b =  
    new Fraction(1, 2);
```



Pass-by-value with Reference Types

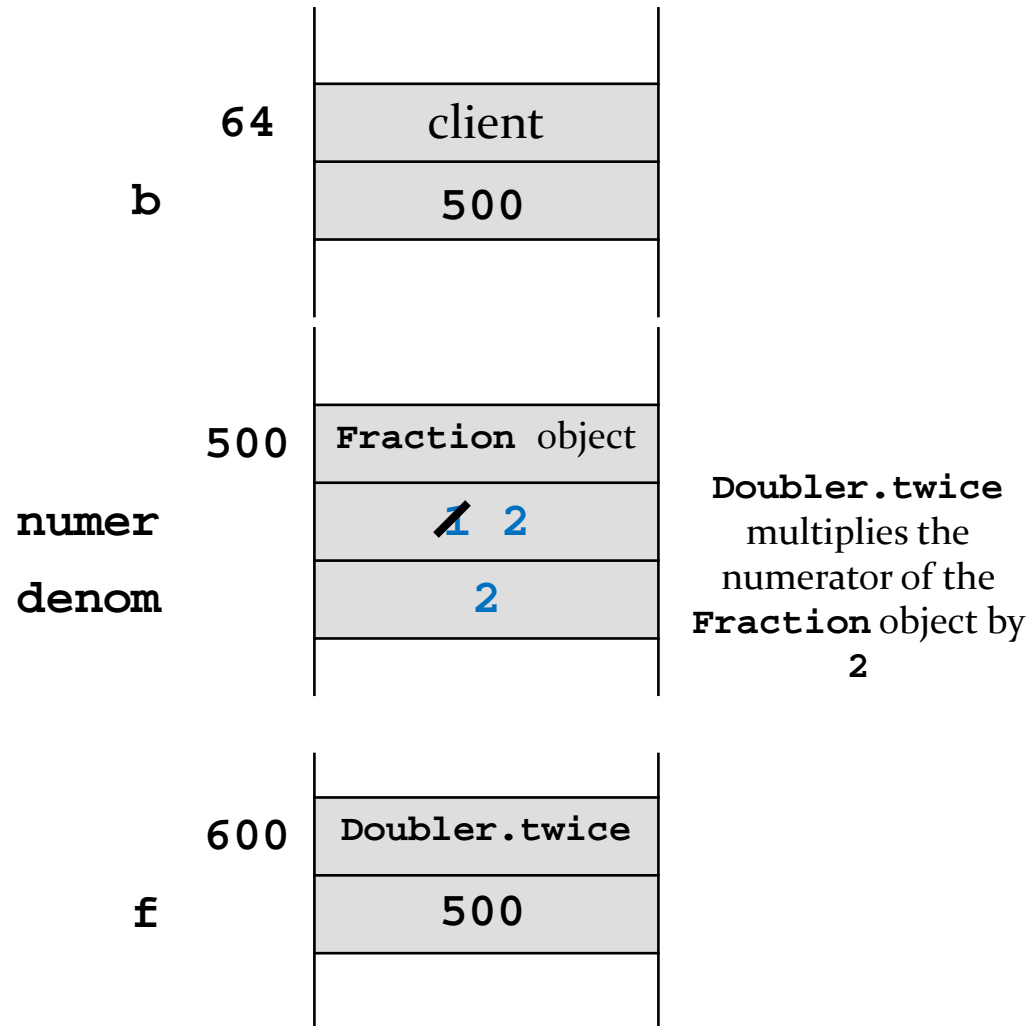
```
Fraction b =  
    new Fraction(1, 2);  
Doubler.twice(b);
```

parameter **f**
is an independent
copy of the value
of argument **b**
(a reference)



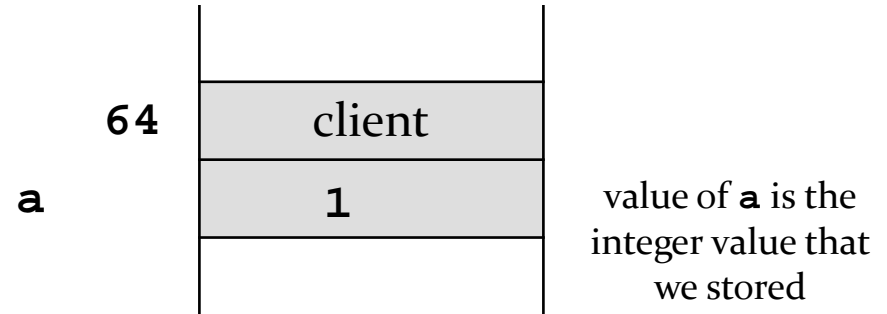
Pass-by-value with Reference Types

```
Fraction b =  
    new Fraction(1, 2);  
Doubler.twice(b);
```



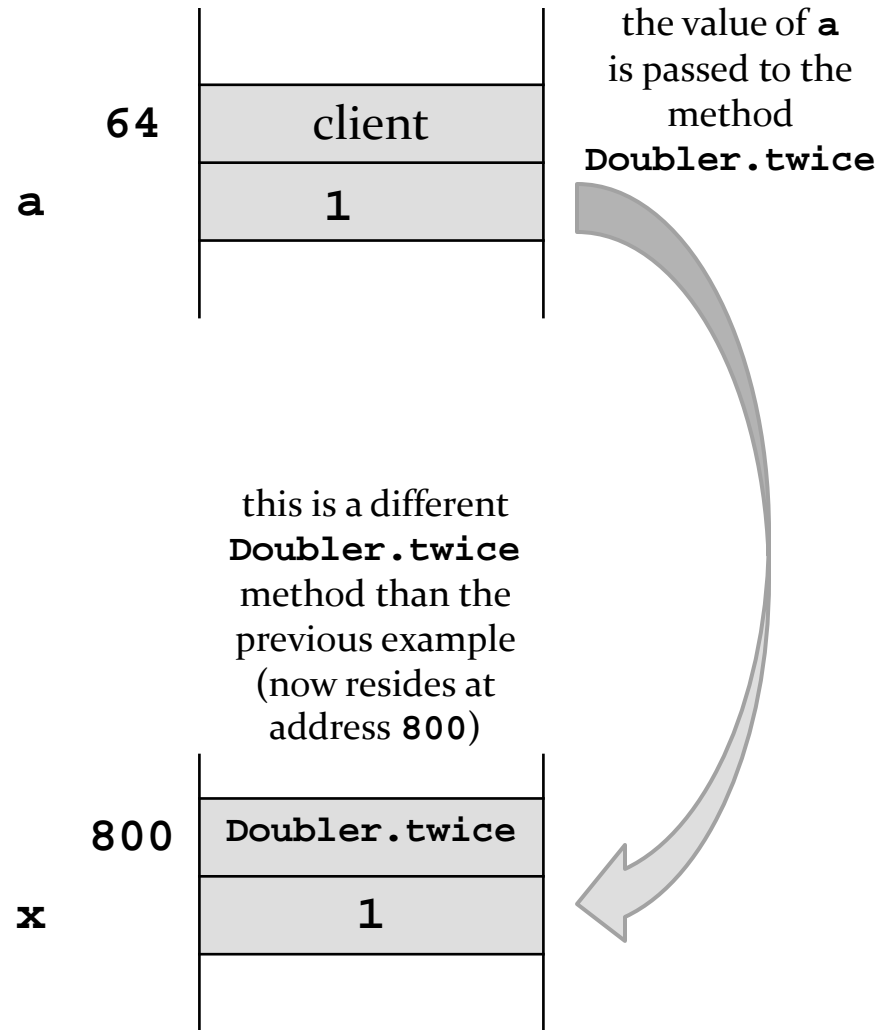
Pass-by-value with Primitive Types

```
int a = 1;
```



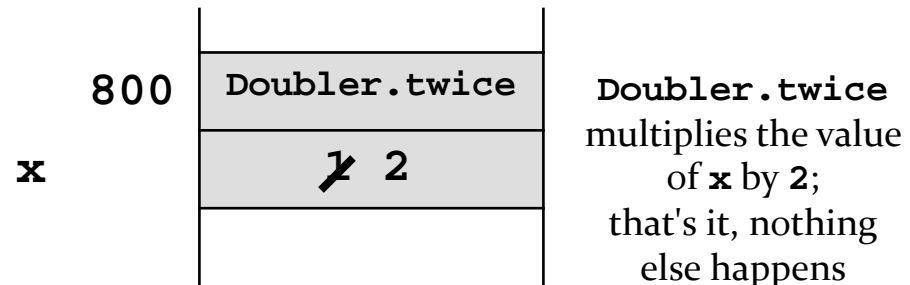
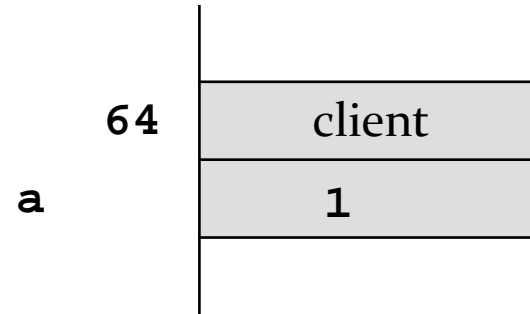
Pass-by-value with Primitive Types

```
int a = 1;  
Doubler.twice(a);
```



Pass-by-value with Reference Types

```
int a = 1;  
Doubler.twice(a);
```



Pass-by-value

- ▶ Java uses pass-by-value for *all* types (primitive and reference)
 - ▶ an argument of primitive type cannot be changed by a method
 - ▶ an argument of reference type can have its state changed by a method
- ▶ pass-by-value is used to return a value from a method back to the client

Documenting Code

Javadoc

- ▶ 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

Javadoc

- ▶ 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`, `@exception`)
 - `@pre`. is non-standard (custom tag used in CSE1030)

Javadoc Guidelines

- ▶ <http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>
- ▶ [notes 1.5.1, 1.5.2]
- ▶ precede every exported class, interface, constructor, method, and attribute with a doc comment
- ▶ for methods the doc comment should describe the contract between the method and the client
 - ▶ preconditions ([notes 1.4], [JBA 2.3.3])
 - ▶ postconditions ([notes 1.4], [JBA 2.3.3])

Javadoc Examples

- ▶ short in-class demo here
- ▶ see any lab exercise

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

- ▶ 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

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

JUnit

- ▶ JUnit is a testing framework for Java
- ▶ A framework is a semi-complete application. A framework provides a reusable, common structure to share among applications. Developers incorporate the framework into their own application and extend it to meet their specific needs"
 - ▶ from the book JUnit in Action

JUnit

- ▶ JUnit provides a way for creating:
 - ▶ test cases
 - ▶ a class that contains one or more tests
 - ▶ test suites
 - ▶ a group of tests
 - ▶ test runner
 - ▶ a way to automatically run test suites
- ▶ in-class demo of JUnit in eclipse

```
package cse1030.games;

import static org.junit.Assert.*;

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

import org.junit.Test;

public class YahtzeeTest {

    @Test
    public void threeOfAKind() {
        // 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));
    }
}
```