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

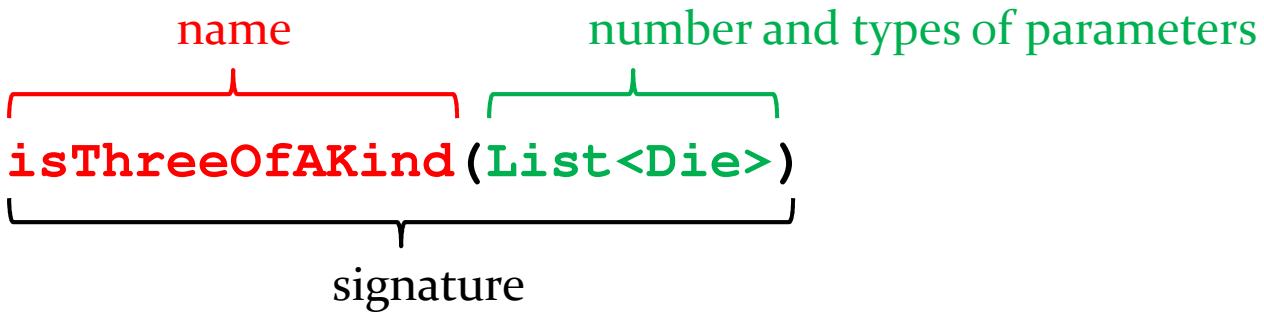
- in Java, a *utility* class is a class having only static fields and static methods
- uses:
  - group related methods on primitive values or arrays
    - **`java.lang.Math`** or **`java.util.Arrays`**
  - group static methods for objects that implement an interface
    - **`java.util.Collections`**
    - [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], [AJ 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());
}
boolean hasTriple = Yahtzee.isThreeOfAKind(dice);
```

argument

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);
  }

}
```

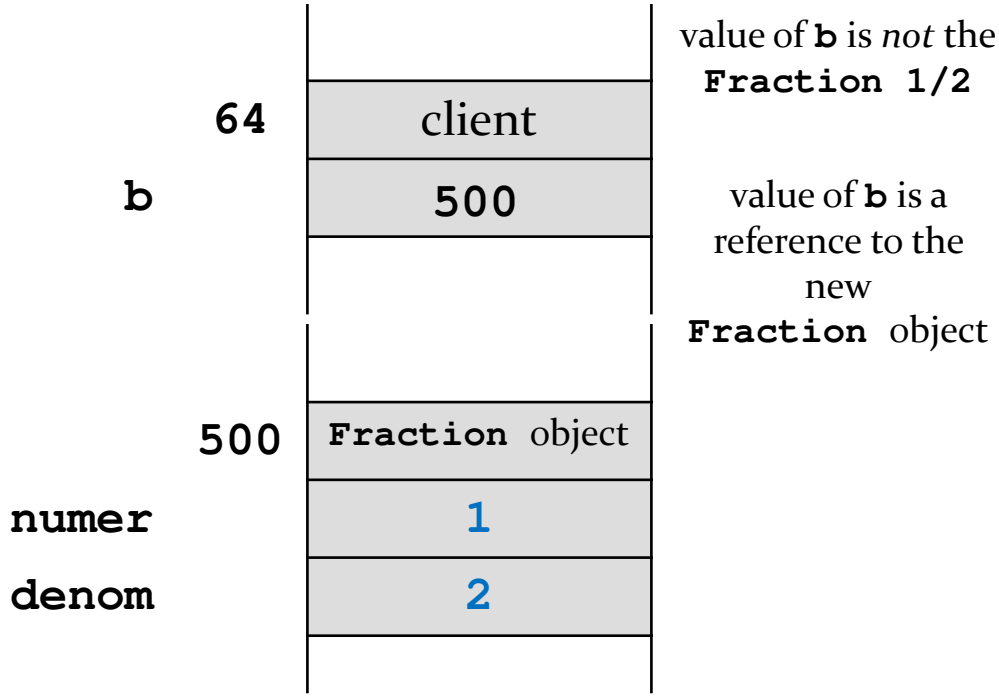
Pass-by-value

- 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

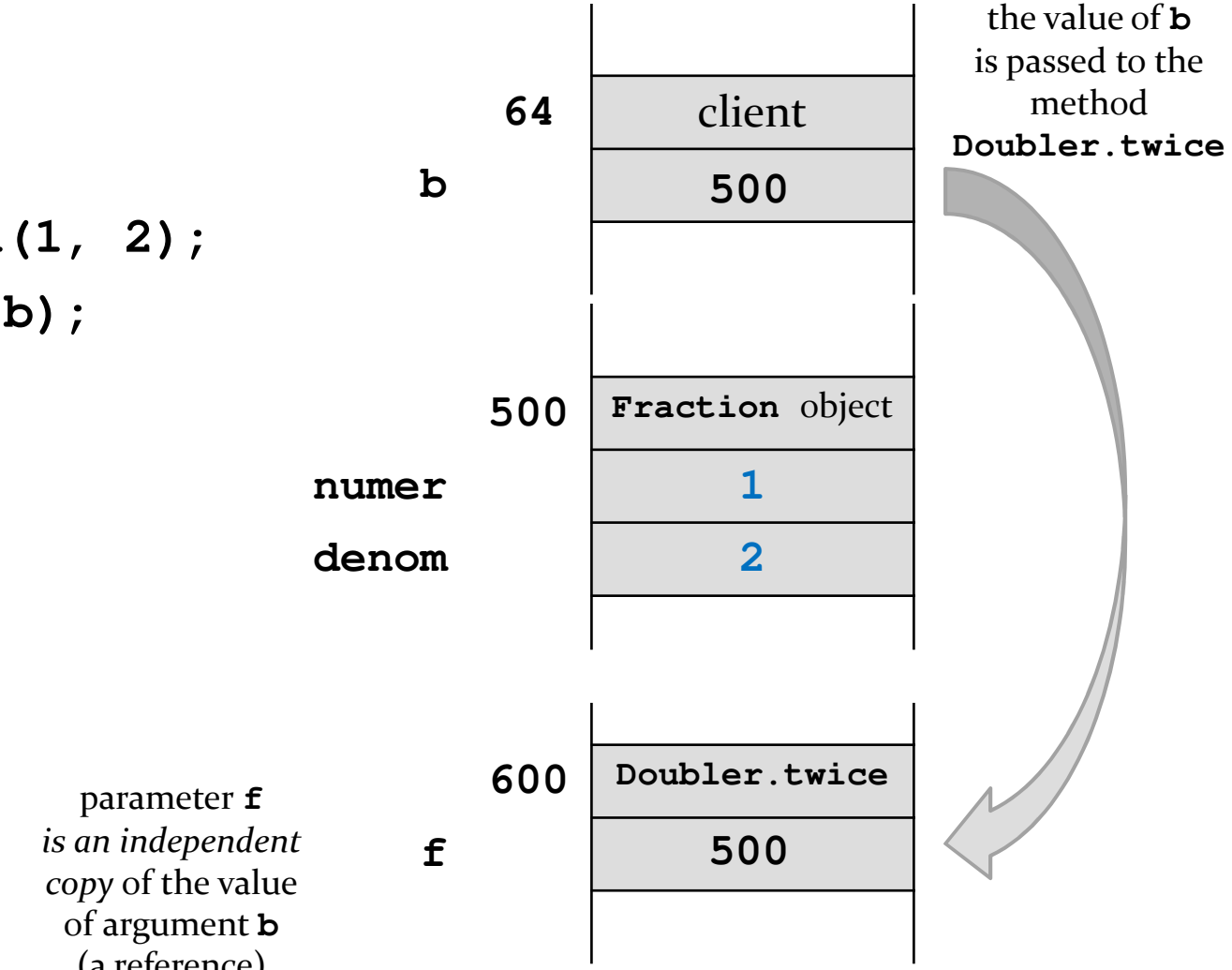
Pass-by-value with Reference Types

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



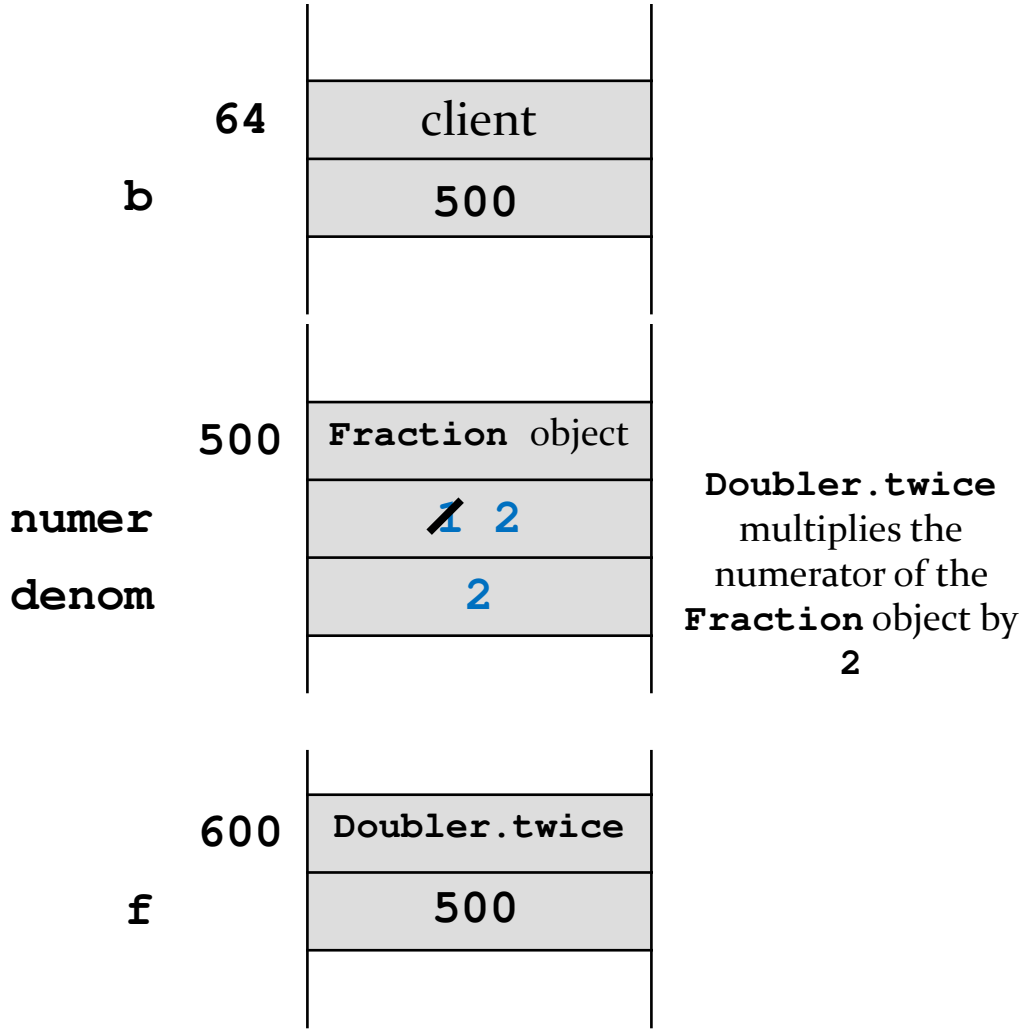
Pass-by-value with Reference Types

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



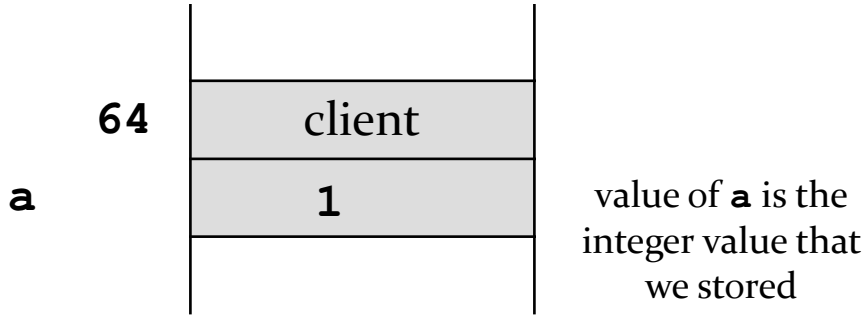
Pass-by-value with Reference Types

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



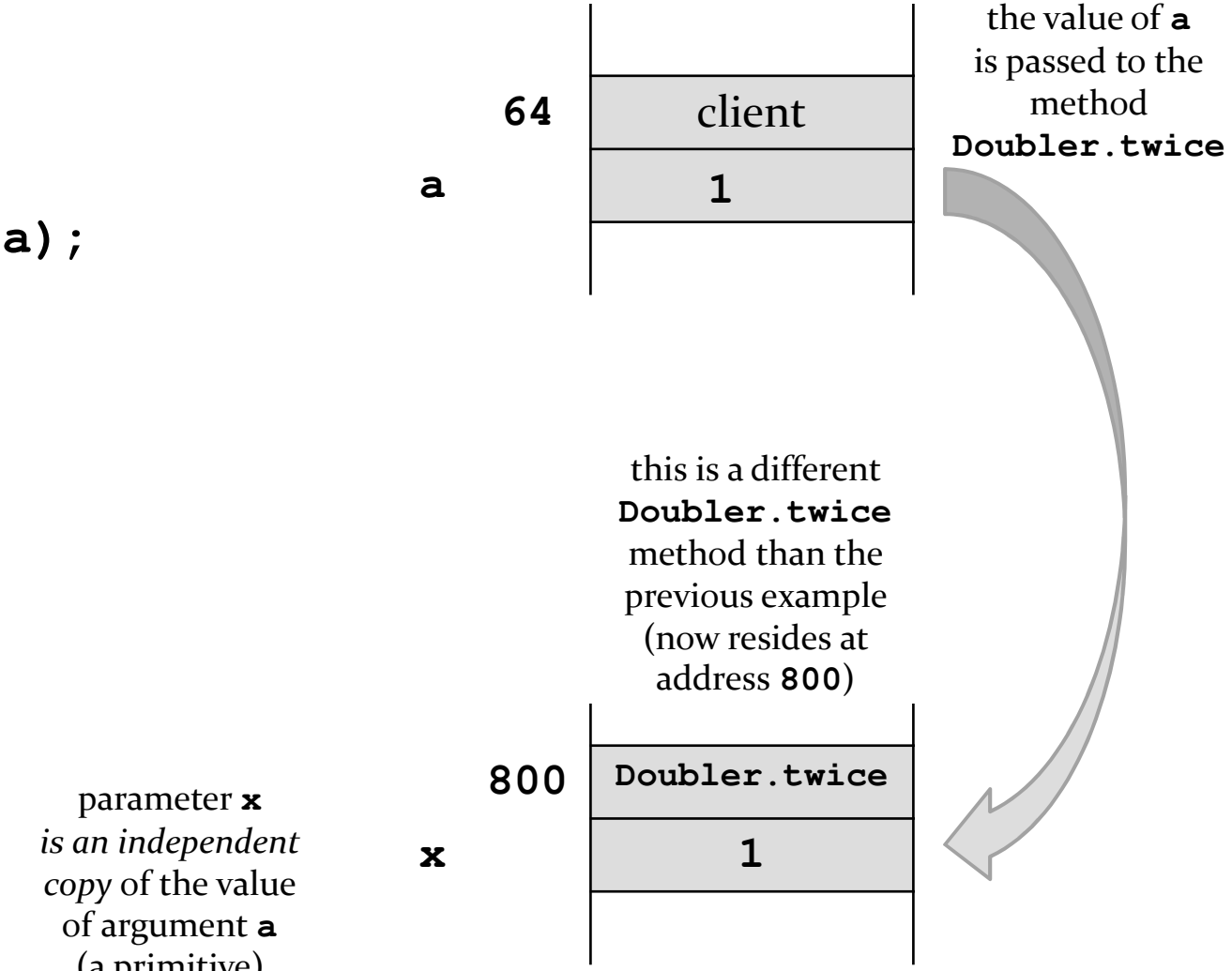
Pass-by-value with Primitive Types

```
int a = 1;
```



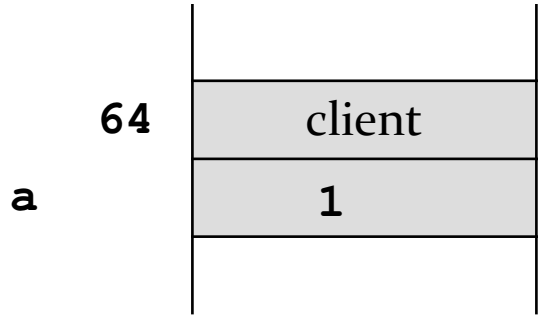
Pass-by-value with Primitive Types

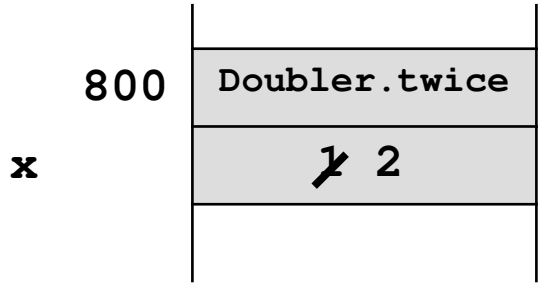
```
int a = 1;
Doubler.twice(a);
```



Pass-by-value with Reference Types

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





`Doubler.twice` multiplies the value of **`x`** by **`2`**; that's it, nothing else happens

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

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

```java
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 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?

```
@Test
public void isThreeOfAKind() {
  for (int i = 1; i <= 6; i++) {
    Die d1 = new Die(6, i);
    Die d2 = new Die(6, i);
    Die d3 = new Die(6, i);
    for (int j = 1; j <= 6; j++) {
      Die d4 = new Die(6, j);
      for (int k = 1; k <= 6; k++) {
        Die d5 = new Die(6, k);
        List<Die> dice = new ArrayList<Die>();
        dice.add(d1);
        dice.add(d2);
        dice.add(d3);
        dice.add(d4);
        dice.add(d5);
        Collections.shuffle(dice);
        assertTrue(Yahtzee.isThreeOfAKind(dice));
      }
    }
  }
}
```

JUnit

- how many variations of three-of-a-kind are tested in our new test?
- how many ways can you roll three-of-a-kind using five dice?

JUnit

- we are now somewhat confident that our method returns **`true`** if the list contains a three-of-a-kind
- but we still have not tested if our method returns **`false`** if the list does not contain a three-of-a-kind

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

```
@Test
public void notThreeOfAKind() {
  final int TRIALS = 1000;
  for (int t = 0; t < TRIALS; t++) {
    List<Die> twelveDice = new ArrayList<Die>();
    for (int i = 1; i <= 6; i++) {
      twelveDice.add(new Die(6, i));
      twelveDice.add(new Die(6, i));
    }
    Collections.shuffle(twelveDice);
    List<Die> dice = twelveDice.subList(0, 5);
    assertFalse(Yahtzee.isThreeOfAKind(dice));
  }
}
```

Explanation of Previous Slide

- a trick is to create a list of 12 dice where there are:
  - 2 ones,
  - 2 twos,
  - 2 threes,
  - 2 fours,
  - 2 fives, and
  - 2 sixes
- shuffle the list (so that the dice appear in some random order)
- use the first 5 dice

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

- See public APIs
- Lab exercises…

Classes (Part 1)

Implementing non-static features

Goals

- implement a small immutable class with *non-static* attributes and methods
  - recipe for immutability
  - **`this`**
  - **`toString`** method
  - **`equals`** method

Value Type Classes

- a *value type* is a class that represents a value
  - examples of values: name, date, colour, mathematical vector
  - Java examples: **`String`**, **`Date`**, **`Integer`**

- the objects created from a value type class can be:
  - mutable: the state of the object can change
    - **`Date`**
  - immutable: the state of the object is constant once it is created
    - **`String`**, **`Integer`** (and all of the other primitive wrapper classes)

Immutable Classes

- a class defines an immutable type if an instance of the class cannot be modified after it is created
  - each instance has its own constant state
    - more precisely, the externally visible state of each object appears to be constant
  - Java examples: **`String`**, **`Integer`** (and all of the other primitive wrapper classes)
- advantages of immutability versus mutability
  - easier to design, implement, and use
  - can never be put into an inconsistent state after creation

North American Phone Numbers

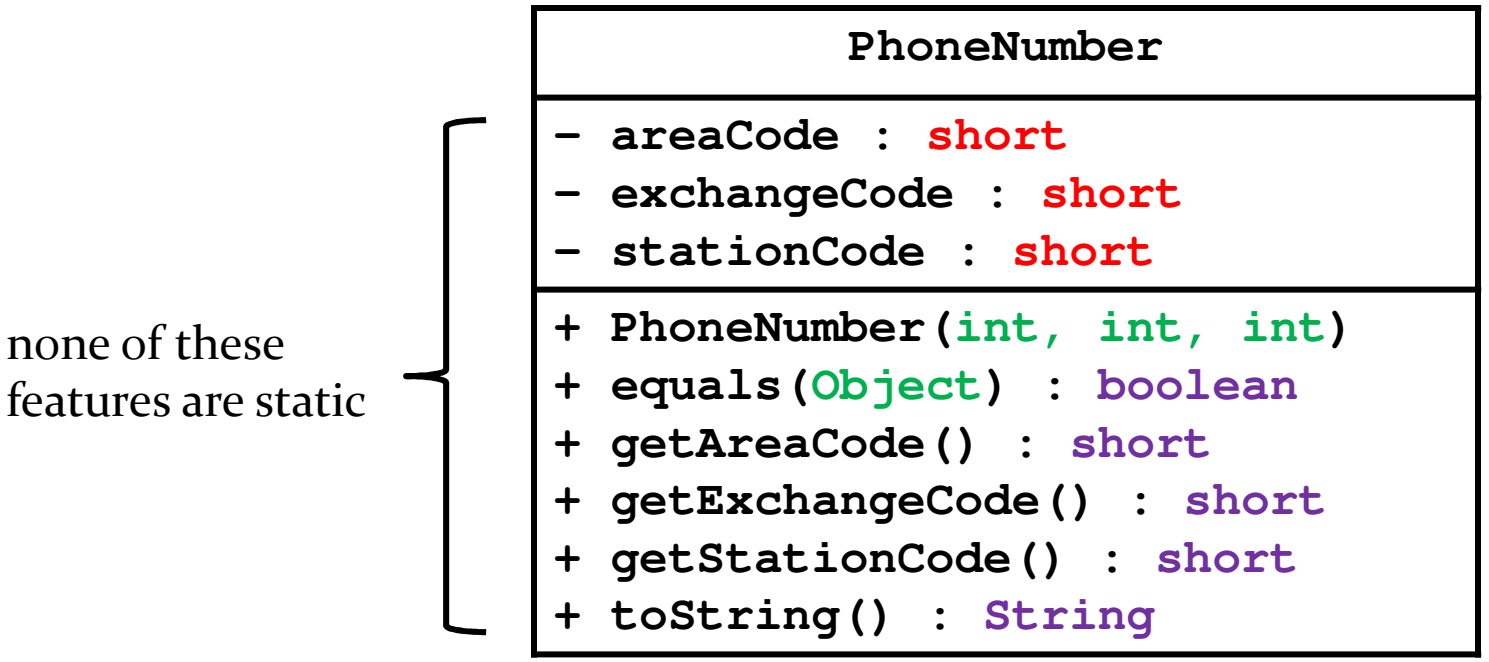
- North American Numbering Plan is the standard used in Canada and the USA for telephone numbers
- telephone numbers look like

416-736-2100

area code	exchange code	station code

Designing a Simple Immutable Class

- **`PhoneNumber`** API



```
package cse1030;

public class PhoneNumber {

}
```

Recipe for Immutability

- the recipe for immutability in Java is described by Joshua Bloch in the book *Effective Java**

1. Do not provide any methods that can alter the state of the object
2. Prevent the class from being extended

revisit when we talk about inheritance

3. Make all fields `final`
4. Make all fields `private`
5. Prevent clients from obtaining a reference to any mutable fields

revisit when we talk about composition

*highly recommended reading if you plan on becoming a Java programmer

Recipe for Immutability 1

1. Do not provide any methods that can alter the state of the object
   - methods that modify state are called *mutators*
   - Java example of a mutator:

```
import java.util.Calendar;

public class CalendarClient {
  public static void main(String[] args)
  {
    Calendar now = Calendar.getInstance();
    // set hour to 5am
    now.set(Calendar.HOUR_OF_DAY, 5);
  }
}
```

Recipe for Immutability 2

2. Prevent the class from being extended
   - one way to do this is to mark the class as **`final`**

- a **`final`** class cannot be extended using inheritance
  - don't confuse **`final`** variable and **`final`** classes

- the reason for this step will become clear in a couple of weeks

```
package cse1030;

public final class PhoneNumber {

}
```

Recipe for Immutability 3

3. Make all fields **`final`**
   - recall that **`final`** means that the field can only be assigned to once
   - **`final`** fields make your intent clear that the class is immutable

```
package cse1030;

public final class PhoneNumber {
  final int areaCode;
  final int exchangeCode;
  final int stationCode;

}
```

Recipe for Immutability 4

4. Make all fields **`private`**
   - this applies to all **`public`** classes (including mutable classes)
   - in **`public`** classes, strongly prefer **`private`** fields
     - and avoid using **`public`** fields
   - **`private`** fields support encapsulation
     - because they are not part of the API, you can change them (even remove them) without affecting any clients
     - the class controls what happens to **`private`** fields
       - it can prevent the fields from being modified to an inconsistent state

```
package cse1030;

public final class PhoneNumber {
  private final int areaCode;
  private final int exchangeCode;
  private final int stationCode;

}
```

Recipe for Immutability 5

5. Prevent clients from obtaining a reference to any mutable fields

- recall that **`final`** fields have constant state only if the type of the attribute is a primitive or is immutable
- if you allow a client to get a reference to a mutable field, the client can change the state of the field, and hence, the state of your immutable class
- revisit this point when we talk about composition
  - also, none of our fields are reference types so we don't have to worry about this point

`this`

- every non-static method of a class has an implicit parameter called **`this`**
- recall that a non-static method requires an object to call the method

```
// client of PhoneNumber

PhoneNumber num = new PhoneNumber(416, 736, 2100);
int areaCode = num.getAreaCode();     // get the
                                      // area code that
                                      // belongs to num
```

- inside **`getAreaCode`**, **`this`** is a reference to object used to invoke the method

getAreaCode

- how does the method **`getAreaCode()`** get the area code for the correct instance?
  - **`this`** is a reference to the calling object

```
/**
 * Get the area code of this phone number.
 *
 * @return the area code of this phone number
 */
public int getAreaCode() {
  return this.areaCode;
}
```

return the area code belonging to the **`PhoneNumber`** object that was used to invoke the method

getExchangeCode and getStationCode

- **`getExchangeCode()`** and **`getStationCode()`** are very similar

```
/**
 * Get the exchange code of this phone number.
 *
 * @return the exchange code of this phone number
 */
public int getExchangeCode() {
  return this.exchangeCode;
}
```

return the exchange code belonging to the **`PhoneNumber`** object that was used to invoke the method

getExchangeCode and getStationCode

- **`getExchangeCode()`** and **`getStationCode()`** are very similar

```
/**
 * Get the station code of this phone number.
 *
 * @return the station code of this phone number
 */
public int getStationCode() {
  return this.stationCode;
}
```

return the station code belonging to the **`PhoneNumber`** object that was used to invoke the method

`toString()`

- recall that every class extends **`java.lang.Object`**
- **`Object`** defines a method **`toString()`** that returns a **`String`** representation of the calling object
  - we can call **`toString()`** with our current **`PhoneNumber`** class

```
// client of PhoneNumber

PhoneNumber num = new PhoneNumber(416, 736, 2100);
System.out.println(num.toString());
```

  - this prints something like
    `phonenumber.PhoneNumber@19821f`

`toString()`

- **`toString()`** should return a concise but informative representation that is easy for a person to read
- it is recommended that all subclasses override this method
  - this means that any non-utility class you write should redefine the **`toString()`** method
    - in this case, our new **`toString()`** method has the same declaration as **`toString()`** in **`java.lang.Object`**

`toString()`

- it is "easy" to override `toString()` for our class

```
/**
 * Returns a string representation of this phone number. The string starts
 * with the area code inside of parenthesis, followed by a space, followed by
 * the exchange code, followed by a hyphen, followed by the station code. The
 * area code and exchange code always have three digits (zero-padded), and the
 * station code always has four digits (zero-padded). For example, the string
 * representation of the phone number 416-736-2100 is:
 *
 * <p>
 * <code>(416) 736-2100</code>
 *
 * @return a string representation of this phone number
 * @see java.lang.Object#toString()
 */
@Override
public String toString() {
  return String.format("(%1$03d) %2$03d-%3$04d",
                       this.areaCode,
                       this.exchangeCode,
                       this.stationCode);
}
```

Constructors

Constructors

- constructors are responsible for initializing instances of a class
  - usually, a constructor will set the fields of the object to:
    - some reasonable default values, or
    - some client specified values,
    - or some combination of the two

[notes 2.2.3]

Constructors

- a constructor declaration looks a little bit like a method declaration:
  - the name of a constructor is the same as the class name
  - a constructor may have an access modifier (but no other modifiers)

```
public PhoneNumber() {

}
```

the *default* constructor (has no parameters)

```
public PhoneNumber(int areaCode,
                   int exchangeCode,
                   int stationCode) {

}
```

a constructor with three parameters

Constructors

- every constructor has an implicit **`this`** parameter
  - the **`this`** parameter is a reference to the object that is currently being constructed

```
public PhoneNumber() {
  this.areaCode = 800;
  this.exchangeCode = 555;
  this.stationCode = 1111;
}

public PhoneNumber(int areaCode,
                   int exchangeCode, int stationCode) {
  this.areaCode = areaCode;
  this.exchangeCode = exchangeCode;
  this.stationCode = stationCode;
}
```

Bell Canada operator phone number?

client specified phone number

Constructors

- a constructor will often need to validate its arguments
  - because you generally should avoid creating objects with invalid state

- what are valid area codes, exchange codes, and station codes?
  - we will assume:
    - must not be negative
    - area code and exchange codes < 1,000
    - station code < 10,000
  - reality is more complicated…

```
public PhoneNumber(int areaCode,
                   int exchangeCode, int stationCode) {

  if (areaCode < 0 || areaCode > 999) {
    throw new IllegalArgumentException("bad area code");
  }
  if (exchangeCode < 0 || exchangeCode > 999) {
    throw new IllegalArgumentException("bad exchange code");
  }
  if (stationCode < 0 || stationCode > 9999) {
    throw new IllegalArgumentException("bad station code");
  }
  this.areaCode = areaCode;
  this.exchangeCode = exchangeCode;
  this.stationCode = stationCode;
}
```

Comment on Immutability

- notice that our constructors make it impossible for a client to create an invalid phone number
- also recall that our class is immutable
  - i.e., the client cannot change a phone number once it is created
- the above two features guarantee that all **`PhoneNumber`** objects will be valid phone numbers