CMSC 131: Chapter 9: Supplement
Classes II

JUnit Testing

JUnit: Utility for the development of software tests.

Setting up JUnit Testing:
- Each test method must start with “test”.
- For now we will learn how to run JUnit tests in Eclipse.
- Later on we will see how to create them.

JUnit Testing (Cont.)

```java
public class AuxMath {
    public static int maximum(int x, int y) {
        if (x > y)
            return x;
        return y;
    }

    public static int minimum(int x, int y) {
        if (x < y)
            return x;
        return y;
    }
}

import junit.framework.TestCase;

public class JUnitTestExample extends TestCase {
    public void testOneMaximum() {
        int expectedResults = 20;
        assertEquals(expectedResults, AuxMath.maximum(10, 20));
    }

    public void testTwoMinimum() {
        int expectedResults = 5;
        assertEquals(expectedResults, AuxMath.minimum(30, 5));
    }
}

JUnit Testing (Cont.)

Execution of Unit test:
Run -> Run As -> JUnit Test

Results:
Green Bar - All test generated expected results.
Brown Bar - Some tests failed (JUnit Failures).
To see more details double click on JUnit tab.
Constructors (revisited)

Constructor: a method that initializes the data for an object.

- It is automatically invoked, whenever a new object instance is created using "new".
- It is important in guaranteeing that an object’s initial state is valid.

```java
public Date( int m, int d, int y ) {
    month = m; day = d; year = y;
    if ( m < 1 || m > 12 ) {
        System.out.println( "Error: month is out of range" );
        System.exit( 0 );
    }
}
```

- The term "constructor" is misleading; think of it as an "initializer".

Constructors (continued)

Constructor Elements:

- It has the same name as the class.
- It has no return type. (But it is not declared as void.)

```java
public Date( int m, int d, int y )     // okay: Date constructor
public void Date( int m, int d, int y )     // no: a method named "Date"
```

- It can be overloaded.
- It can call other methods, but other methods generally cannot call it.

Visibility: Constructors can be public or private, but they are usually public.

Default Constructor

No-Argument Constructor:

A constructor with no arguments (sometimes called a default constructor). It is a good idea to provide such a constructor, since a class user may not have a good initial setting for class known.

Java’s default constructor:

If you do not provide any constructor, Java provides one that sets:

- all numeric instance variables to 0,
- all boolean instance variables to false, and
- all references instance variables to null.
But, if you provide even one constructor (even if it has arguments) Java provides no default constructor.

**Copy Constructor**

**Copy Constructor:** Initializes this object to be a copy of another.

```java
public Rational( Rational r ) { // copy constructor
    if ( r == null ) { // cannot initialize from null
        System.out.println("Illegal construction from null reference");
        System.exit( 0 );
    }
    set( r.numer, r.denom );
}
```

**Notes:**
- Always check that the object being copied is non-null.
- Copying is important to avoid aliasing.

<table>
<thead>
<tr>
<th>Rational q = r;</th>
<th>// bad! q is an alias to r (one instance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rational q = new Rational( r );</td>
<td>// good: creates a new instance</td>
</tr>
</tbody>
</table>

- Java recommends that instead of copy constructors, you use a method called clone. (We will discuss this later, but use copy constructors until then.)

**Classes and Objects**

**Review of Objects/Classes:**

**Variable:** Stores either a primitive type or an object reference.

**Objects:** Any object consists of:
- State:
- Behavior:

Each individual object you create in your program is called an object instance.

**Class:** A definition (blueprint) for an object. A class consists of:
- **Instance variables:** (also called fields)
- **Class methods:**

new: (unlike primitive types) object instances are created by the "new" operator, which allocates space in the heap.

**Reference:**

**Visibility:**
Instance Variables

Review of Instance Variables:

Variables come in two basic types:

Instance variables: sit within a class, but not within any method.

Local variables: (are not instance variables) They sit within a particular method.

Visibility: (private, public) is the same as for methods.

Static/Non-static Instance Variables:

Non-Static (the default): associated with a single instance.

Static: not associated with any one instance, but are shared by all instances.

Methods

Review of Methods:

Methods define an object’s behavior

Method visibility:

public: Accessible from both inside and outside the class
private: Accessible from only inside the class

Method types: Each method returns a value of a specified type or no value (void).

Parameters:

Static/Non-static Methods:

Non-Static (the default): associated with a single instance

Static: are not associated with any one instance, but are shared by all.

Overloading: Having the same name, but different parameter types.
Example: Rational

Let us consider a class `Rational`, which implements a rational number as a fraction:

```
    numerator / denominator
```

Both quantities are integers. We want our class to support methods for:

- **Initializing** a new rational number (**constructors**)
- **Converting** a rational number to a **string** (for printing)
- **Accessing** and **modifying** the value of the number
- **Comparing** two rational number for equality
- **Performing** basic rational **operations** (reciprocal, multiply, etc.)

**Constructors for Rational**

**Instance variables:**

```
    Numerator: int numer;
    Denominator: int denom;
```

**Constructors:**

- **No-argument (default) constructor:**
- **Standard constructor:**
- **Integer-valued constructor:**
- **Copy constructor:**

To simplify their implementation, we define a private utility

```
    set( int n, int d )
```

which sets the value of the numerator and denominator.
Example: Constructors for Rational

```java
public class Rational {
    private int numer; // numerator
    private int denom; // denominator

    private void set(int n, int d) { numer = n; denom = d; }
    public Rational() { set(0, 1); }
    public Rational(int n, int d) { set(n, d); }
    public Rational(int n) { set(n, 1); }

    public Rational(Rational r) {
        if (r == null) { // cannot initialize from null
            System.out.println("Illegal construction from null reference");
            System.exit(0);
        }
        set(r.numer, r.denom);
    }
    // ... (rest of class omitted for now)
}
```

Example: Using Rational Constructors

```
Rational r0 = new Rational();
Rational r1 = new Rational(2, 5);
Rational r2 = new Rational(4, 10);
Rational r3 = r1;
Rational r4 = new Rational(r1);
Rational r5 = null;
```

Memory map:

**Accessors and Mutators**

Since class instances variables are usually private, it is common to define special methods to read or modify their values.

**Rational public accessor**s: (for reading)
- `int getNumerator()`: returns value of numerator
- `int getDenominator()`: returns value of denominator
- `double doubleValue()`: returns number value as a double
- `double floatValue()`: returns number value as a float

**Rational public mutator**s: (for modifying)
- `void setNumerator(int n)`: set the numerator
- `void setDenominator(int d)`: set the denominator
- `void setValue(int v)`: set the entire value to v (i.e., v/1)
Example: Accessors/Mutators for Rational

public class Rational {

    // ... (instance variables and constructors omitted) ...

    public String toString() {
        String result = numerator + "/" + denominator + "(" + doubleValue() + ");
        return result;
    }

    public int getNumerator() { return numerator; }
    public int getDenominator() { return denominator; }
    public double doubleValue() { return (double) numerator / (double) denominator; }
    public float floatValue() { return (float) numerator / (float) denominator; }

    public void setNumerator( int n ) { numerator = n; }
    public void setDenominator( int d ) { denominator = d; }
    public void setValue( int v ) { set( v, 1 ); }

    // ... (rest of class omitted for now) ...
}

Example: Accessors/Mutators for Rational

public static void main( String[] args ) {

    Rational r0;
    r0.setNumerator( 3 );

    Rational r1 = null;
    r1.setNumerator( 3 );

    Rational r2 = new Rational();
    r2.setNumerator( 2 );
    r2.setDenominator( 3 );

    System.out.println( r2 );
    System.out.println( r2.toString() );

    System.out.println( "float = " + r2.floatValue() );
    System.out.println( "double = " + r2.doubleValue() );

    r2.setDenominator( r2.getNumerator() );
}
Multiplication for Rationals

Multiplication: Want to multiply Rationals q and r.

Static multiplication of two arguments: Returns a new rational:

    Rational p = Rational.multiply( q, r ); // p = q*r

This method will call new to create a new Rational number and return its reference.

Nonstatic equivalent of the above: Returns a new rational:

    Rational p = q.multiply( r );       // p = q*r

This behaves the same as the above method. But the style is little uglier, because it lacks
symmetry. (We won't do this.)

Nonstatic multiplication, which modifies q:

    q.multiplyBy( r );                 // q *= r

No new object is created, and q is modified as a result. It is okay for this to be asymmetric,
since only q is modified.

Example: Multiplication for Rationals

```java
public class Rational {
    // ... (previous stuff omitted)...

    public static Rational multiply( Rational q, Rational r )
    { return new Rational( q.numer * r.numer, q.denom * r.denom ); }

    public void multiplyBy( Rational r )
    { numer *= r.numer; denom *= r.denom; }

    // ... (rest of class omitted for now)...
}

public static void main( String[] args ){
    Rational s1 = new Rational( -3, 8 );
    Rational s2 = new Rational( 2 );
    Rational s3 = Rational.multiply( s1, s2 );
    System.out.println( s1 );
    System.out.println( s2 );
    System.out.println( s3 );
    s1.multiplyBy( s2 );
    System.out.println( s1 );
}
```
Example: Equality Tests for Rational

```java
public class Rational {
    // ... (other stuff omitted)...

    public static boolean equals( Rational q, Rational r )
    { return q.numer * r.denom == r.numer * q.denom; }

    public boolean equals( Rational r )
    { return equals( this, r ); }

    public static boolean identical( Rational q, Rational r )
    { return q.numer == r.numer && q.denom == r.denom; }
}

public static void main( String[] args )
{  Rational t1 = new Rational( 2, -4 );
    if ( t1 == -0.5 ) ...
    if ( t1 == Rational( -1, 2 ) ) ...
    if ( t1 == new Rational( -1, 2 ) ) ...
    if ( t1.equals( new Rational( -1, 2 ) ) ) ...
    Rational t2 = new Rational( -3, 6 );
    if ( Rational.equals( t1, t2 ) ) ...
    if ( Rational.identical( t1, t2 ) ) ...
}
```