P2: Exercise 1 Discussion

Claudio Corrodi

2 March 2018

Two approaches

- Custom algorithm
  - Recursive
  - Look at the first character of pattern and filename at a time
- Regular expressions
  - One-liner can cover most cases
  - But: What about special character?

Custom Algorithm using recursion

```
private boolean match_rec(String pattern, String filename) {
  ...
  if (pattern.charAt(0) != filename.charAt(0)) {
    return false;
  } else {
    return match_rec(pattern.substring(1), filename.substring(1));
  }
  ...
}
```

Custom Algorithm using recursion

```
private boolean match_rec(String pattern, String filename) {
  ...
  if (pattern.charAt(0) != filename.charAt(0)) {
    return false;
  } else {
    return match_rec(pattern.substring(1), filename.substring(1));
  }
  ...
}
```

match_rec(“abc”, “abcde.txt”) ==
match_rec(“bc”, “bcde.txt”) ==
match_rec(“c”, “cde.txt”) ==
match_rec(“”, “de.txt”) == ...

Custom Algorithm using recursion

```
private boolean match_rec(String pattern, String filename) {
  ...
   // Question mark. If filename is not empty, match the remainder
   // of pattern to the remainder of filename.
   if (pattern.startsWith("?")) {
      if (filename.isEmpty()) {
         return false;
      } else {
         return match_rec(pattern.substring(1),
         ↪  filename.substring(1));
      }
   }
 }
```

Custom Algorithm using recursion

```
private boolean match_rec(String pattern, String filename) {
  ...
   // Question mark. If filename is not empty, match the remainder
   // of pattern to the remainder of filename.
   if (pattern.startsWith("?")) {
      if (filename.isEmpty()) {
         return false;
      } else {
         return match_rec(pattern.substring(1),
         ↪   filename.substring(1));
      }
   }
 }
```

match_rec(“?oo.txt”, “foo.txt”) ==
match_rec(“oo.txt”, “oo.txt”) == ...

Custom Algorithm using recursion

```
private boolean match_rec(String pattern, String filename) {
  ...
   // Star. Try to match any remainder.
  for (int i = 0; i <= filename.length(); i++) {
    if (match_rec(pattern.substring(1), filename.substring(i))) {
      return true;
    }
  }
  return false;
}
```

Custom Algorithm using recursion

```
private boolean match_rec(String pattern, String filename) {
  ...
   // Star. Try to match any remainder.
  for (int i = 0; i <= filename.length(); i++) {
    if (match_rec(pattern.substring(1), filename.substring(i))) {
      return true;
    }
  }
  return fals
}
```

match_rec("*.txt", "foo.txt") ==
match_rec(".txt", "foo.txt") $\vee$
match_rec(".txt", "oo.txt") $\vee$
match_rec(".txt", "o.txt") $\vee$
match_rec(".txt", ".txt")

Regular expressions

```
private boolean matchRegex(String filename) {
   String regexPattern = pattern;
   regexPattern = regexPattern.replace("*", ".*");
   regexPattern = regexPattern.replace("?", ".");
   return Pattern.matches(regexPattern, filename);
}
```

"." matches exactly one character
".*" matches any number of characters

Regular expressions

```
private boolean matchRegex(String filename) {
   String regexPattern = pattern;
   regexPattern = regexPattern.replace("*", ".*");
   regexPattern = regexPattern.replace("?", ".");
   return Pattern.matches(regexPattern, filename);
}
```

"." matches exactly one character
".*" matches any number of characters

- What about special characters?
  ⇒ Read the documentation!

```
regexPattern = regexPattern.replace(".", "\\.");
```

Examples: Encapsulation & names

```
public class FilePattern {

  public String string;

  public FilePattern(String string) {
    this.string = string;
  }
}
```

Examples: Encapsulation & names

```
public class FilePattern {

  public String string;

  public FilePattern(String string) {
    this.string = string;
  }
}
```

Examples: Encapsulation & names

Make attributes protected

Use meaningful names

```
pu                                        {

  protected String pattern

  public FilePattern(String pattern) {
    this.string = string;
  }
}
```

Examples: Useless code

```
protected String tempPattern;

public String getTempPattern() {
    return this.tempPattern;
}
```

Examples: Useless code

```java
protected String tempPattern;

public String getTempPattern() {
    return this.tempPattern;
}
```

Unused outside of class! Use `tempPattern` directly.

Manual Testing

```
public class TestMain {
  public static void main(String[] args) {
    FilePattern a = new FilePattern("fname*");
    System.out.println(a.matches("fname.txt"));
  }
}
```

Manual Testing

```
public class TestMain {
  public static void main(String[] args) {
    FilePattern a = new FilePattern("fname*");
    System.out.println(a.matches("fname.txt"));
  }
}
```

```
public class FilePatternTest {
  @Test
  public void fnameStarMatchesFnameDotTxt() {
    FilePattern a = new FilePattern("fname*");
    assertTrue(a.matches("fname.txt"));
  }
}
```

Add the scenario as a **permanent** test

Javadoc

Alex Syrel

P2

Java supports three comment types:

```
/**
 * A documentation comment.
 */

/*
 * A standard comment.
 */

// A one-line comment.
```

Java supports three comment types:

```
/**
 * A documentation comment.
 */

/*
 * A standard comment.
 */


// A one-line comment.
```

Why to document?

Code is read much more often than it is written

Even if you don't intend anybody else to read your code, that somebody is probably going to be you, twelve months from now

```java
public Affine2 setToTrnRotScl(float x, float y, float degrees,
float scaleX, float scaleY) {
  m02 = x;
  m12 = y;

  if (degrees == 0) {
    m00 = scaleX;
    m01 = 0;
    m10 = 0;
    m11 = scaleY;
  } else {
    float sin = MathUtils.sinDeg(degrees);
    float cos = MathUtils.cosDeg(degrees);

    m00 = cos * scaleX;
    m01 = -sin * scaleY;
    m10 = sin * scaleX;
    m11 = cos * scaleY;
  }
  return this;
}
```
https://github.com/libgdx/libgdx/blob/master/gdx/src/com/badlogic/gdx/math/Affine2.java

Inform others how to use your code without having to read it

```
/** Sets this matrix to a concatenation of translation, rotation and scale.
 * It is a more efficient   form for:
 * <code>idt().translate(x, y).rotate(degrees).scale(scaleX, scaleY)</code>
 * @param x The translation in x.
 * @param y The translation in y.
 * @param degrees The angle in degrees.
 * @param scaleX The scale in y.
 * @param scaleY The scale in x.
 * @return This matrix for the purpose of chaining operations.
 */
```

```
public Affine2 setToTrnRotScl(
     m02 = x;
     m12 = y;

     if (degrees == 0) {
          m00 = scaleX;
          m01 = 0;
          m10 = 0;
          m11 = scaleY;
     } else
          float
          float

          m00 = cos * scaleX;
          m01 = -sin * scaleY;
          m10 = sin * scaleX;
          m11 = cos * scaleY;
     }
     return
}
```

https://github.com/libgdx/libgdx/blob/master/gdx/src/com/badlogic/gdx/math/Affine2.java

What is Good Documentation?

Make the first sentence count

Javadoc assumes it to be the summary

```
/**
 * When I was a kid I had absolutely no idea
 * the day will come when I stop writing code
 * and begin to do JavaDoc.
 * Nevertheless this method returns 42.
 *
 * @return 42
 */
```

Do not use fillers!

This method/function/class... is not necessary.

```
/**
 * This is a nice method to assert beautiful quality
 * of amazing chars at a given index under the moonlight
 */
```

First word should be a verb

helps to understand code faster

```
/**
 * Removes user from the list
 */

/**
 * Translates window to the left
 */

/**
 * Establishes network connection
 */
```

Remember to describe

corner cases. e.g. null? negative ints?

```
/**
 * ...
 * Moves snake to specified position.
 * Snake should not be null as long as
 * position is positive and less then 10
 * ...
 */
 public void moveTo(int position) { }
```

Would be nice

link to other documentation – with @see or @link

```
/**
 * Convenience for calling {@link Window#getLayoutInflater}.
 *
 * @see android.view.Window
 */
 public LayoutInflater getLayoutInflater() {
    return getWindow().getLayoutInflater();
 }
```

https://github.com/android/platform_frameworks_base/blob/master/core/java/android/app/Activity.java

Class Comments

What is the class responsible for?

What information does it hold?

What things can it do?

Who uses this class?

How should the class be used?

Does this class need special treatment?

```
/**
 * An activity is a single, focused thing that the user can do.  Almost all
 * activities interact with the user, so the Activity class takes care of
 * creating a window for you in which you can place your UI with
 * {@link #setContentView}.  While activities are often presented to the user
 * as full-screen windows, they can also be used in other ways: as floating
 * windows (via a theme with {@link android.R.attr#windowIsFloating} set)
 * or embedded inside of another activity (using {@link ActivityGroup}).
 *
 * There are two methods almost all subclasses of Activity will implement:
 *
 * <ul>
 *     <li> {@link #onCreate} is where you initialize your activity.  Most
 *     importantly, here you will usually call {@link #setContentView(int)}
 *     with a layout resource defining your UI, and using {@link #findViewById}
 *     to retrieve the widgets in that UI that you need to interact with
 *     programmatically.
 *
 *     <li> {@link #onPause} is where you deal with the user leaving your
 *     activity.  Most importantly, any changes made by the user should at this
 *     point be committed (usually to the
 *     {@link android.content.ContentProvider} holding the data).
 * </ul>
 *
 * <p>To be of use with {@link android.content.Context#startActivity Context.startActivity()}, all
 * activity classes must have a corresponding
 * {@link android.R.styleable#AndroidManifestActivity &lt;activity&gt;}
 * declaration in their package's <code>AndroidManifest.xml</code>.</p>
 *
 * <p>Topics covered here:
 * <ol>
 * <li><a href="#Fragments">Fragments</a>
 * <li><a href="#ActivityLifecycle">Activity Lifecycle</a>
 * <li><a href="#ConfigurationChanges">Configuration Changes</a>
 * <li><a href="#StartingActivities">Starting Activities and Getting Results</a>
 * <li><a href="#SavingPersistentState">Saving Persistent State</a>
 * <li><a href="#Permissions">Permissions</a>
 * <li><a href="#ProcessLifecycle">Process Lifecycle</a>
 * </ol>
 *
 * <div class="special reference">
 * <h3>Developer Guides</h3>
 * <p>The Activity class is an important part of an application's overall lifecycle,
 * and the way activities are launched and put together is a fundamental
 * part of the platform's application model. For a detailed perspective on the structure of an
 * Android application and how activities behave, please read the
 * <a href="{@docRoot}guide/topics/fundamentals.html">Application Fundamentals</a> and
 * <a href="{@docRoot}guide/topics/fundamentals/tasks-and-back-stack.html">Tasks and Back Stack</a>
 * developer guides.</p>
 *
 * <p>You can also find a detailed discussion about how to create activities in the
 * <a href="{@docRoot}guide/topics/fundamentals/activities.html">Activities</a>
 * developer guide.</p>
 * </div>
 *
 * <a name="Fragments"></a>
 * <h3>Fragments</h3>
 *
 * <p>Starting with {@link android.os.Build.VERSION_CODES#HONEYCOMB}, Activity
 * implementations can make use of the {@link Fragment} class to better
 * modularize their code, build more sophisticated user interfaces for larger
 * screens, and help scale their application between small and large screens.
 *
 * <a name="ActivityLifecycle"></a>
 * <h3>Activity Lifecycle</h3>
 *
 * <p>Activities in the system are managed as an <em>activity stack</em>.
 * When a new activity is started, it is placed on the top of the stack
 * and becomes the running activity -- the previous activity always remains
 * below it in the stack, and will not come to the foreground again until
 * the new activity exits.</p>
 *
 * <p>An activity has essentially four states:</p>
 * <ul>
 *     <li> If an activity in the foreground of the screen (at the top of
 *         the stack),
 *         it is <em>active</em> or  <em>running</em>. </li>
 *     <li>If an activity has lost focus but is still visible (that is, a new non-full-sized
 *         or transparent activity has focus on top of your activity), it
 *         is <em>paused</em>. A paused activity is completely alive (it
 *         maintains all state and member information and remains attached to
 *         the window manager), but can be killed by the system in extreme
 *         low memory situations.
 *     <li>If an activity is completely obscured by another activity,
 *         it is <em>stopped</em>. It still retains all state and member information,
 *         however, it is no longer visible to the user so its window is hidden
 *         and it will often be killed by the system when memory is needed
 *         elsewhere.</li>
 *     <li>If an activity is paused or stopped, the system can drop the activity
 *         from memory by either asking it to finish, or simply killing its
 *         process.  When it is displayed again to the user, it must be
 *         completely restarted and restored to its previous state.</li>
 * </ul>
 *
 * <p>The following diagram shows the important state paths of an Activity.
 * The square rectangles represent callback methods you can implement to
 * perform operations when the Activity moves between states.  The colored
 * ovals are major states the Activity can be in.</p>
 *
 * <p><img src="../../../images/activity_lifecycle.png"
 *      alt="State diagram for an Android Activity Lifecycle." border="0" /></p>
 *
 * <p>There are three key loops you may be interested in monitoring within your
 * activity:
 *
 * <ul>
 * <li>The <b>entire lifetime</b> of an activity happens between the first call
 * to {@link android.app.Activity#onCreate} through to a single final call
 * to {@link android.app.Activity#onDestroy}.  An activity will do all setup
 * of "global" state in onCreate(), and release all remaining resources in
 * onDestroy().  For example, if it has a thread running in the background
 * to download data from the network, it may create that thread in onCreate()
 * and then stop the thread in onDestroy().
 *
 * <li>The <b>visible lifetime</b> of an activity happens between a call to
 * {@link android.app.Activity#onStart} until a corresponding call to
 * {@link android.app.Activity#onStop}.  During this time the user can see the
 * activity on-screen, though it may not be in the foreground and interacting
 * with the user.  Between these two methods you can maintain resources that
 * are needed to show the activity to the user.  For example, you can register
 * a {@link android.content.BroadcastReceiver} in onStart() to monitor for changes
 * that impact your UI, and unregister it in onStop() when the user no
 * longer sees what you are displaying.  The onStart() and onStop() methods
 * can be called multiple times, as the activity becomes visible and hidden
 * to the user.
 *
 * <li>The <b>foreground lifetime</b> of an activity happens between a call to
 * {@link android.app.Activity#onResume} until a corresponding call to
 * {@link android.app.Activity#onPause}.  During this time the activity is
 * in front of all other activities and interacting with the user.  An activity
 * can frequently go between the resumed and paused states -- for example when
 * the device goes to sleep, when an activity result is delivered, when a new
 * intent is delivered -- so the code in these methods should be fairly
 * lightweight.
 * </ul>
 *
 * <p>The entire lifecycle of an activity is defined by the following
 * Activity methods.  All of these are hooks that you can override
 * to do appropriate work when the activity changes state.  All
 * activities will implement {@link android.app.Activity#onCreate}
 * to do their initial setup; many will also implement
 * {@link android.app.Activity#onPause} to commit changes to data and
 * otherwise prepare to stop interacting with the user.  You should always
 * call up to your superclass when implementing these methods.</p>
 *
 * </p>
 * <pre class="prettyprint">
 * public class Activity extends ApplicationContext {
 *     protected void onCreate(Bundle savedInstanceState);
 *
 *     protected void onStart();
 *
 *     protected void onRestart();
 *
 *     protected void onResume();
 *
 *     protected void onPause();
 *
 *     protected void onStop();
 *
 *     protected void onDestroy();
 * }
 * </pre>
 *
 * <p>In general the movement through an activity's lifecycle looks like
 * this:</p>
 *
 * <table border="2" width="85%" align="center" frame="hsides" rules="rows">
 *     <colgroup align="left" span="3" />
 *     <colgroup align="left" />
 *     <colgroup align="center" />
 *     <colgroup align="center" />
 *
 *     <thead>
 *     <tr><th colspan="3">Method</th> <th>Description</th> <th>Killable?</th> <th>Next</th></tr>
 *     </thead>
 *
 *     <tbody>
 *     <tr><th colspan="3" align="left" border="0">{@link android.app.Activity#onCreate onCreate()}</th>
 *         <td>Called when the activity is first created.
 *             This is where you should do all of your normal static set up:
 *             create views, bind data to lists, etc.  This method also
 *             provides you with a Bundle containing the activity's previously
 *             frozen state, if there was one.
 *             <p>Always followed by <code>onStart()</code>.</td>
 *         <td align="center">No</td>
 *         <td align="center"><code>onStart()</code></td>
 *     </tr>
 *
 *     <tr><td rowspan="5" style="border-left: none; border-right: none;">&nbsp;&nbsp;&nbsp;&nbsp;</td>
 *         <th colspan="2" align="left" border="0">{@link android.app.Activity#onRestart onRestart()}</th>
 *         <td>Called after your activity has been stopped, prior to it being
 *             started again.
 *             <p>Always followed by <code>onStart()</code></td>
 *         <td align="center">No</td>
 *         <td align="center"><code>onStart()</code></td>
 *     </tr>
 *
 *     <tr><th colspan="2" align="left" border="0">{@link android.app.Activity#onStart onStart()}</th>
 *         <td>Called when the activity is becoming visible to the user.
 *             <p>Followed by <code>onResume()</code> if the activity comes
 *             to the foreground, or <code>onStop()</code> if it becomes hidden.</td>
 *         <td align="center">No</td>
 *         <td align="center"><code>onResume()</code> or <code>onStop()</code></td>
 *     </tr>
 *
 *     <tr><td rowspan="2" style="border-left: none;">&nbsp;&nbsp;&nbsp;&nbsp;</td>
 *         <th align="left" border="0">{@link android.app.Activity#onResume onResume()}</th>
 *         <td>Called when the activity will start
 *             interacting with the user.  At this point your activity is at
 *             the top of the activity stack, with user input going to it.
 *             <p>Always followed by <code>onPause()</code>.</td>
 *         <td align="center">No</td>
 *         <td align="center"><code>onPause()</code></td>
 *     </tr>
 *
 *     <tr><th align="left" border="0">{@link android.app.Activity#onPause onPause()}</th>
 *         <td>Called when the system is about to start resuming a previous
 *             activity.  This is typically used to commit unsaved changes to
 *             persistent data, stop animations and other things that may be consuming
 *             CPU, etc.  Implementations of this method must be very quick because
 *             the next activity will not be resumed until this method returns.
 *             <p>Followed by either <code>onResume()</code> if the activity
 *             returns back to the front, or <code>onStop()</code> if it becomes
 *             invisible to the user.</td>
 *         <td align="center"><font color="#800000"><strong>Pre-{@link android.os.Build.VERSION_CODES#HONEYCOMB}</strong></font></td>
 *         <td align="center"><code>onResume()</code> or<br>
 *                 <code>onStop()</code></td>
 *     </tr>
 *
 *     <tr><th colspan="2" align="left" border="0">{@link android.app.Activity#onStop onStop()}</th>
 *         <td>Called when the activity is no longer visible to the user, because
 *             another activity has been resumed and is covering this one.  This
 *             may happen either because a new activity is being started, an existing
 *             one is being brought in front of this one, or this one is being
 *             destroyed.
 *             <p>Followed by either <code>onRestart()</code> if
 *             this activity is coming back to interact with the user, or
 *             <code>onDestroy()</code> if this activity is going away.</td>
 *         <td align="center"><font color="#800000"><strong>Yes</strong></font></td>
 *         <td align="center"><code>onRestart()</code> or<br>
 *                 <code>onDestroy()</code></td>
 *     </tr>
 *
 *     <tr><th colspan="3" align="left" border="0">{@link android.app.Activity#onDestroy onDestroy()}</th>
 *         <td>The final call you receive before your
 *             activity is destroyed.  This can happen either because the
 *             activity is finishing (someone called {@link Activity#finish} on
 *             it, or because the system is temporarily destroying this
 *             instance of the activity to save space.  You can distinguish
 *             between these two scenarios with the {@link
 *             Activity#isFinishing} method.</td>
 *         <td align="center"><font color="#800000"><strong>Yes</strong></font></td>

 * <p>Note the "Killable" column in the above table -- for those methods that
 * are marked as being killable, after that method returns the process hosting the
 * activity may killed by the system <em>at any time</em> without another line
 * of its code being executed.  Because of this, you should use the
 * {@link #onPause} method to write any persistent data (such as user edits)
 * to storage.  In addition, the method
 * {@link #onSaveInstanceState(Bundle)} is called before placing the activity
 * in such a background state, allowing you to save away any dynamic instance
 * state in your activity into the given Bundle, to be later received in
 * {@link #onCreate} if the activity needs to be re-created.
 * See the <a href="#ProcessLifecycle">Process Lifecycle</a>
 * section for more information on how the lifecycle of a process is tied
 * to the activities it is hosting.  Note that it is important to save
 * persistent data in {@link #onPause} instead of {@link #onSaveInstanceState}
 * because the latter is not part of the lifecycle callbacks, so will not
 * be called in every situation as described in its documentation.</p>
 *
 * <p class="note">Be aware that these semantics will change slightly between
 * applications targeting platforms starting with {@link android.os.Build.VERSION_CODES#HONEYCOMB}
 * vs. those targeting prior platforms.  Starting with Honeycomb, an application
 * is not in the killable state until its {@link #onStop} has returned.  This
 * impacts when {@link #onSaveInstanceState(Bundle)} may be called (it may be
 * safely called after {@link #onPause()} and allows and application to safely
 * wait until {@link #onStop()} to save persistent state.</p>
 *
 * <p>For those methods that are not marked as being killable, the activity's
 * process will not be killed by the system starting from the time the method
 * is called and continuing after it returns.  Thus an activity is in the killable
 * state, for example, between after <code>onPause()</code> to the start of
 * <code>onResume()</code>.</p>
 *
 * <a name="ConfigurationChanges"></a>
 * <h3>Configuration Changes</h3>
 *
 * <p>If the configuration of the device (as defined by the
 * {@link Configuration Resources.Configuration} class) changes,
 * then anything displaying a user interface will need to update to match that
 * configuration.  Because Activity is the primary mechanism for interacting
 * with the user, it includes special support for handling configuration
 * changes.</p>
 *
 * <p>Unless you specify otherwise, a configuration change (such as a change
 * in screen orientation, language, input devices, etc) will cause your
 * current activity to be <em>destroyed</em>, going through the normal activity
 * lifecycle process of {@link #onPause},
 * {@link #onStop}, and {@link #onDestroy} as appropriate.  If the activity
 * had been in the foreground or visible to the user, once {@link #onDestroy} is
 * called in that instance then a new instance of the activity will be
 * created, with whatever savedInstanceState the previous instance had generated
 * from {@link #onSaveInstanceState}.</p>
 *
 * <p>This is done because any application resource,
 * including layout files, can change based on any configuration value.  Thus
 * the only safe way to handle a configuration change is to re-retrieve all
 * resources, including layouts, drawables, and strings.  Because activities
 * must already know how to save their state and re-create themselves from
 * that state, this is a convenient way to have an activity restart itself
 * with a new configuration.</p>
 *
 * <p>In some special cases, you may want to bypass restarting of your
 * activity based on one or more types of configuration changes.  This is
 * done with the {@link android.R.attr#configChanges android:configChanges}
 * attribute in its manifest.  For any types of configuration changes you say
 * that you handle there, you will receive a call to your current activity's
 * {@link #onConfigurationChanged} method instead of being restarted.  If
 * a configuration change involves any that you do not handle, however, the
 * activity will still be restarted and {@link #onConfigurationChanged}
 * will not be called.</p>
 *
 * <a name="StartingActivities"></a>
 * <h3>Starting Activities and Getting Results</h3>
 *
 * <p>The {@link android.app.Activity#startActivity}
 * method is used to start a
 * new activity, which will be placed at the top of the activity stack.  It
 * takes a single argument, an {@link android.content.Intent Intent},
 * which describes the activity
 * to be executed.</p>
 *
 * <p>Sometimes you want to get a result back from an activity when it
 * ends.  For example, you may start an activity that lets the user pick
 * a person in a list of contacts; when it ends, it returns the person
 * that was selected.  To do this, you call the
 * {@link android.app.Activity#startActivityForResult(Intent, int)}
 * version with a second integer parameter identifying the call.  The result
 * will come back through your {@link android.app.Activity#onActivityResult}
 * method.</p>
 *
 * <p>When an activity exits, it can call
 * {@link android.app.Activity#setResult(int)}
 * to return data back to its parent.  It must always supply a result code,
 * which can be the standard results RESULT_CANCELED, RESULT_OK, or any
 * custom values starting at RESULT_FIRST_USER.  In addition, it can optionally
 * return back an Intent containing any additional data it wants.  All of this
 * information appears back on the
 * parent's <code>Activity.onActivityResult()</code>, along with the integer
 * identifier it originally supplied.</p>
 *
 * <p>If a child activity fails for any reason (such as crashing), the parent
 * activity will receive a result with the code RESULT_CANCELED.</p>
 *
 * <pre class="prettyprint">
 * public class MyActivity extends Activity {
 *     ...
 *
 *     static final int PICK_CONTACT_REQUEST = 0;
 *
 *     public boolean onKeyDown(int keyCode, KeyEvent event) {
 *         if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
 *             // When the user center presses, let them pick a contact.
 *             startActivityForResult(
 *                 new Intent(Intent.ACTION_PICK,
 *                 new Uri("content://contacts")),
 *                 PICK_CONTACT_REQUEST);
 *            return true;
 *         }
 *         return false;
 *     }
 *
 *     protected void onActivityResult(int requestCode, int resultCode,
 *             Intent data) {
 *         if (requestCode == PICK_CONTACT_REQUEST) {
 *             if (resultCode == RESULT_OK) {
 *                 // A contact was picked.  Here we will just display it
 *                 // to the user.
 *                 startActivity(new Intent(Intent.ACTION_VIEW, data));
 *             }
```

Method Comments

Remember to describe

Parameters (@param)

```
/**
 * Throws an appropriate exception based
 * on the passed in error code.
 *
 * @param code - the DND error code,
 *               should be positive
 */
 public static void error (int code) {
   error (code, 0);
 }
```

https://github.com/eclipse/eclipse.platform.swt/blob/master/bundles/org.eclipse.swt/Eclipse%20SWT%20Drag%20and%20Drop/common/org/eclipse/swt/dnd/DND.java

Remember to describe

Return (@return)

```
/**
 * Get the source of this exception event.
 *
 * @return The {@link Throwable} that is
 *      the source of this exception event.
 */
public Throwable getException() {
  return (Throwable) getSource();
}
```

Remember to describe

Exceptions (@throws)

```
/**
 * ...
 * @throws android.content.ActivityNotFoundException
 *  if there was no Activity found to run the given Intent.
 * ...
 */
public void startActivityForResult(Intent intent, int requestCode)
 throws ActivityNotFoundException {
    startActivityForResult(intent, requestCode, null);
}
```

https://github.com/android/platform_frameworks_base/blob/master/core/java/android/app/Activity.java

Examples

```java
public class ServerProxy implements IServer {
    /* ... */
}
```

```java
/**
 * Relays method calls to a remote { @see Server }.
 * <p>
 * The proxy is responsible for establishing and
 * keeping a connection to the server. The caller
 * must ensure that a connection is destroyed with
 * the {@see #disconnect} method.
 */
 public class ServerProxy implements IServer {
     /* ... */
 }
```

```java
/**
 * Constructor
 */
 public ServerProxy(String url , int port)
       throws NetworkConnectionException {
   /* ... */
 }
```

```java
/**
 * Establishes a connection to a remote server.
 * Throws if it fails to do so.
 *
 * @param url address that can either be resolved
 *      via hosts.conf or DNS or is an IP address.
 *
 * @param port port to connect to on the server. A
 *    positive integer , typically above 1024.
 *    Must be the same as the {@see Server}
 *    uses with its {@see Server#listenOn} method.
 *
 * @throws NetworkConnectionException if it was
 *           not able to initiate a connection.
 */
public ServerProxy(String url , int port)
      throws NetworkConnectionException {
  /* ... */
}
```

```
/**
 * Ends the connection
 */
public void disconnect () {
  // ...
}
```

```
/**
 * Returns the number of jobs
 */
public int getJobCount () {
  // ...
}
```

```java
/**
 * Ends the connection. After this call, no other
 * method call is valid, including this one. The
 * server is not affected by this.
 */
public void disconnect () {
  // ...
}

/**
 * Returns the number of jobs running on the server.
 *
 * @return a non-negative integer that is the
 *      number of jobs that are alive.
 */
public int getJobCount () {
  // ...
}
```

```java
/**
 * Returns the url of the server.
 */
public String getUrl () {
  return  url;
}
```

```
public String getUrl () {
  return  url;
}
```

Sometimes no comments are best comments

```
/**
 * The end
 */
```

Exercise 2

Exercise 2: Snakes & Ladders

- You are given a skeleton for the Snakes & Ladders game
- Add new types of squares
- Test behaviour of squares (using JUnit)
- Write proper documentation

Junit

- Testing framework
  - Covered in more detail in lecture 4
- Goal: Make sure program behaves as expected
- **JUnit**: Individual, independent tests

JUnit

```
@Test
public void newGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    Player[] args = { jack, jill };
    Game game = new Game(12, args);
    game.setSquareToLadder(2, 4);
    game.setSquareToLadder(7, 2);
    game.setSquareToSnake(11, -6);
    assertTrue(game.notOver());
    assertTrue(game.firstSquare().isOccupied());
    assertEquals(1, jack.position());
}
```

JUnit

```
@Test
public void newGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    Player[] args = { jack, jill };
    Game game = new Game(12, args);
    game.setSquareToLadder(2, 4);
    game.setSquareToLadder(7, 2);
    game.setSquareToSnake(11, -6);
    assertTrue(game.notOver());
    assertTrue(game.firstSquare().isOccupied());
    assertEquals(1, jack.position());
}
```

Initialize player

JUnit

```
@Test
public void newGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    Player[] args = { jack, jill };
    Game game = new Game(12, args);
    game.setSquareToLadder(2, 4);
    game.setSquareToLadder(7, 2);
    game.setSquareToSnake(11, -6);
    assertTrue(game.notOver());
    assertTrue(game.firstSquare().isOccupied());
    assertEquals(1, jack.position());
}
```

Specify expected output

JUnit

```
@Test
public void newGame() {
    jack = new Player("Jack");
    jill = new Player("Jill");
    Player
    Game ga
    game.se
    game.se
    game.se
    assertT
    assertTrue(game.firstSquare().isOccupied());
    assertEquals(1, jack.position());
}
```

The exercise comes with some existing tests for reference.

More in exercise_02.md
git pull p2-exercises master