Contents

0 Introduction

We are going to do a simple case study of creating a very simple text editor; it will look like the familiar `NotePad` application that is the default text editor on certain computers.

Woven into this exposition will be an introduction to file IO and to Java's exception handling mechanism. You will get to see this mechanism in action right away. We will create an app that is capable of creating text files, opening exsiting text files, editing them, and saving to them.

Here are some features we should have.

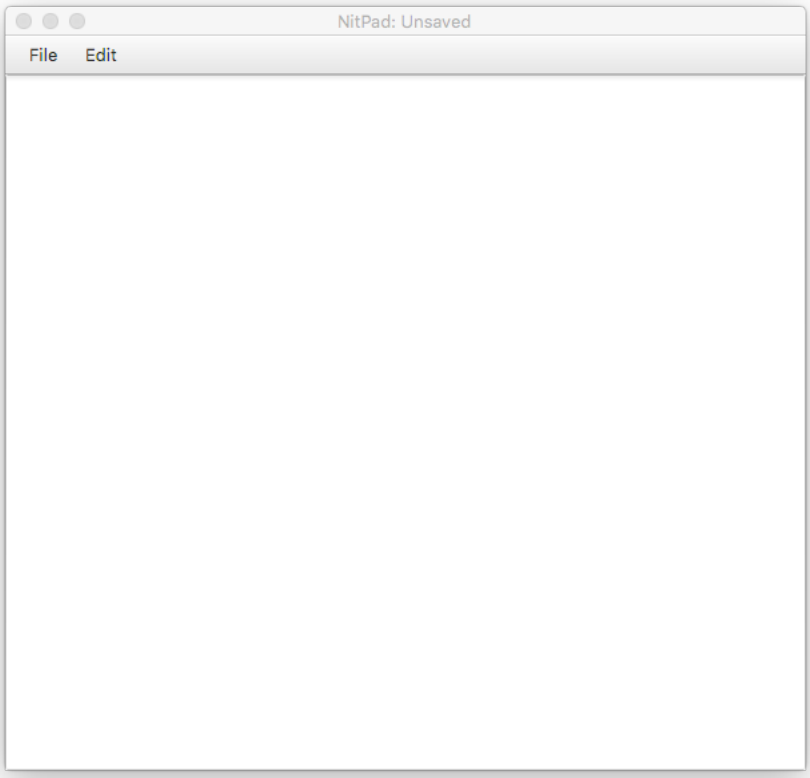
1. We should have a `File` menu, with the standard items including New, Open, Save, Save As and Quit.
2. We need areas for typing text. We will introduce the `TabPane`, which allows us to have several work areas open at once.
3. We will put the absolute path of the currently opened file in a status bar, or mark the status bar with `Unsaved File`.
4. We should have a `Edit` menu, with the standard items including Copy, Paste, Cut, and Select All.
5. We will develop all this in such a manner that the user will be protected from losing data.

Let us begin by creating the menus.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuItem;
import javafx.scene.control.MenuBar;
import javafx.scene.control.TabPane;
import javafx.scene.layout.BorderPane;
public class NitPad extends Application
{
    private TabPane tp;
    private BorderPane bp;
    private MenuBar mbar;
    public static void main(String[] args)
    {
        launch(args);
    }
    @Override
    public void init()
    {
        tp = new TabPane();
        bp = new BorderPane();
        mbar = new MenuBar();
    }
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("NitPad");
        bp.setTop(mbar);
```

```
        bp.setCenter(tp);
        createFileMenu();
        createEditMenu();
        primary.setScene(new Scene(bp, 600,550));
        primary.show();
    }
    private void createFileMenu()
    {
        Menu fileMenu = new Menu("File");
        mbar.getMenus().add(fileMenu);
    }
    private void createEditMenu()
    {
        Menu editMenu = new Menu("Edit");
        mbar.getMenus().add(editMenu);
    }
}
```

Compile and run and see this.



Note we made several items state variable so they will be visible in more than one method. This allows us to keep the `start` method from becoming unmanageably big. We will now go on a brief detour and learn what we need about files, so we can read from them, write to them, and manage them.

1 The File Class and Paths

During this chapter we will become acquainted with a new package, `java.io`, which deals with file input/output operations, or file IO, for short.

The File class belongs to this package; it is used to represent locations in your file system. The File class does not play a role in the actual reading or writing of data to a file. Instances of the `File` class can point at files or directories stored on your system. There are other objects that handle the actual mechanics of file IO.

Let us explore this class and see what it does. Open a DrJava interactive session, and the Java API guide. Let us begin by looking at the Field Summary in the guide. It features four fields. Really, they harbor two pieces of data, the path separator and the separator character. This character can be yielded up as a character or a string. Hence the existence of four constants. Notice that these constants are static, so we can call them by class name.

```
> import java.io.File;
> File.pathSeparator
":"
> File.pathSeparatorChar
':'
> File.separator
"/"
> File.separatorChar
'/'
>
```

The purpose of the separator character is to separate files in a path. This character is a on Windoze systems, and it is a / on a UNIX system. The notion of path is common to all operating systems. Recall a path consists of a sequence of directories followed by a directory or file. The separator character can be expanded to "and then into." Only the last item in a path can be a regular file; all others must be directories.

For example the path

```
animals/mammals/tapir.html
```

specifies a file tapir.html that lives inside of directory mammals, which in turn lives inside of directory animals. In Windoze, this path is specified by

```
animals\mammals\tapir.html
```

Common to the command line interfaces of Windoze and UNIX is the notion of `search path`. In UNIX if you enter the command `ls`, it not in your `cwd`. Therefore UNIX checks your search path, which is a list of paths to directories for the presence of `ls`. It checks this list in order; if it finds the command in some directory, it immediately executes it. Since `ls` lives in `/bin`, the directory `/bin` must be in your path for `ls` to run. Fortunately, this is done for you by default. Windoze also has a search path mechanism that works in an identical way. Let us show the path on both systems. First on UNIX, we see the path by entering `echo $PATH`at the command prompt.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$
```

In Windoze, bring up a console window by going to the run menu item in the Start menu, and type `cmd` into the text slot. Then, in this little black window, type `PATH` at the prompt.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\John Morrison>PATH
PATH=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
```

In both systems, there is an environment variable containing the search path. In UNIX this is `$PATH`, in Windoze it is `PATH`. You will also notice that the search path is a list of absolute paths. Absolute paths in Windoze begin with a drive letter such as `C:`; in UNIX they begin with a `/`. Observe that the directory `/bin` is listed on the path of this machine, so ls there is found and run. If you type an command that does not live in your path, you will get an error message, as seen here.

```
$ ontv
bash: ontv: command not found
$
```

We sent UNIX on a wild goose chase and we get rewarded with a nastygram. If you are running a MAC you will see the same things. In Windoze, you will see this, informing us it has been sent on a similarly futile wild goose chase.

```
C:\Documents and Settings\John Morrison>ontv
âĂŹontvâĂŹ is not recognized as an internal or external command,
operable program or batch file.
```

We see that the path separator, which is : in UNIX and ; in Windoze, separates directories in the search path.

2 Constructors and Methods

We shall most commonly use the constructor

```
public File(String pathname)
```

to create instances of `File` objects. Such an object may point at a directory or a regular file. This is not surprising to UNIX users, since they know that directories are just special files that contain an index to their contents. You may use an absolute path name, or a relative path name. A relative path name will

be relative to the `cwd` of the java program when it is running on the userâĂŹs machine.

The `exists()` method can be used to see if a given file already exists on the userâĂŹs system. Here we show a brief example.

```
> f = new File(".");
> f.exists()
true
> g = new File("someFileThatDoesNotExist")
> g.exists()
false
>
```

We began by making f point at the Java programâĂŹs `cwd`. Naturally, this must exist. We then deliberately chose a file that does not exist in the programâĂŹs cwd, and we see that exists() discerns its nonexistence. This method can be very useful for performing file IO; you can use it to avoid clobbering an existing (valuable) file on the (hapless) userâĂŹs system.

The `canRead()`, `canWrite()`, and `canExecute()` methods are self-explanatory. They come in handy: you can check to see if your program has‘ permission for gaining access to a file prior to charging forth. This can save the throwing of an exception.

The methods `getPath()` and `getAbsolutePath()` will return the string representation of the path to a file from the program's `cwd`. You should make a directory, place a few unvaluable files in it, and experiment with the methods in the API. You can remove files, make new directories, and do all manner of file management with this class. Do the simple exercises below in a program or in an interactive DrJava session.

Programming Exercises

1. Make a new `File` object and use it to determine the absolute path of your `cwd`.
2. Perform ls -l on your cwd. Try resetting permission bits on one of your files by creating a file object. Verify what you did using ls -l, and by using methods from the `File` class.
3. Make a `File` object in the interactive prompt and change its `cwd` to various places. Check for existence and nonexistence of various files. See if you can determine what permissions you have for the files. Can you check if a file is a directory?
4. See if you can write a program that takes a file or directory as command line argument and which imitates the action of the UNIX command `ls`.

3 What are Exceptions?

This next major component of the Java language will allow us to write programs that handle errors gracefully. Java provides a mechanism called *exception handling* that provides a parallel track of return from functions so that you can avoid cluttering the ordinary execution of code with endless error-handling routines.

We have seen these already. No doubt you have run code and seen an one of these occur and the subsequent ugly death of your program. Consider this unfortunate code.

```
import java.util.ArrayList;
public class DumbError
{
    private static ArrayList<String> roster;

    public static void main(String[] args)
    {
        roster.add("Doofus McDuff");
        System.out.println(roster);
    }
}
```

This code compiles.

```
$ javac DumbError.java
```

Now let us run it.

```
$ java DumbError
Exception in thread "main" java.lang.NullPointerException
    at DumbError.main(DumbError.java:8)
$
```

Doom. You see that a `NullPointerException` got thrown. You have no doubt seen this before. The compiler did not catch the error; an exception is a run-time error. You will see that on line 8 the offending code is

```
    roster.add("Doofus McDuff");
```

This occured because we never did this

```
    roster = new ArrayList<>();
```

We attempted to call a method on an object pointing to the graveyard state `null`. Insert this line we just showed and your program will run without error. Other exceptions you probably have run across include these.

- `IndexOutOfBoundsException` This is thrown when you attempt to index into an array or array list with a nonexistent index.
- `StringIndexOutOfBoundsException` This is thrown when you attempt to index into a string using `charAt()` using a nonexistent index.

Exceptions are objects that are "thrown" by various methods or actions. In this chapter we will learn how to handle (catch) an exception. By so doing we allow our program to recover and continue to work. Failure to catch and exception results in a flood of nasty red text from Java (a so-called "exploding heart"). Crashes such as these should be extremely rare in production-quality software. We can use exceptions as a means to protect our program from such dangers as user abuse and from such misfortunes as crashing whilst attempting to gain access to an nonexistent or prohibited resource. Many of these hazards are beyond both user and programmer control.

When you program with files or with socket connections, the handling of exceptions will be mandatory; hence the need for this chapter before we begin handling files.

Let us now turn to understanding how exceptions fit into Java's class hierarchy.

Progrmming Exercise

1. What happens if you try to compute 1/0 using integers?
2. What happens if you try to compute 1/0 using doubles?
3. What happens if you make the call `Math.sqrt(-1)`?
4. What happens if you make the call `Math.log(-1)`?

4 The `Throwable` Subtree

Go to the Java API guide and pull up the class `Exception`. The family tree is is as follows.

```
java.lang.Object
java.lang.Throwable
java.lang.Exception
```

The class name `Throwable` is a bit strange; one would initially think it were an interface. It is, however, a class. The class `java.lang.Exception` has a sibling class `java.lang.Error`.

When objects of type `Error` are thrown, it is not reasonable to try to recover. These things come from problems in the Java Virtual Machine, bad memory problems, or problems from the underlying OS. We just accept the fact that they cause program death. Continuing to proceed would just lead to a chain of ever-escalating problems.

Objects of type `Exception` are thrown for more minor problems, such as an attempt to open a non-existent file for reading, trying to convert an unparseable string to an integer, or trying to access an entry of a string, array, or array list that is out of bounds.

Let us show this mechanism at work. For example, if you attempt to execute the code

```
int foo = Integer.parseInt("gobbledegook");
```

you will be rewarded with an exception. To see what happens, create this program `MakeException.java`.

```
public class MakeException
{
    public static void main(String[] args)
    {
        int foo = Integer.parseInt("gobbledegook");
    }
}
```

This program compiles happily. You will see that the infraction we have here is a run-time error, as is any exception.

When you run the program you will see this in the interactions pane.

```
java.lang.NumberFormatException: For input string: "gobbledegook"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at MakeException.main(MakeException.java:5)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at edu.rice.cs.drjava.model.compiler.JavacCompiler.runCommand(JavacCompiler.java:272)
```

The exploding heart you see here shows a *stack trace*. This shows how the exception propagates through the various calls the program makes. To learn what you did wrong, you must look in this list for your file(s). You will see the offending line here.

```
at MakeException.main(MakeException.java:5)
```

You are being told that the scene of the crime is on line 5, smack in the middle of your `main` method. The stack trace can yield valuable clues in tracking down and extirpating a run-time problem such as this one.

We have seen reference to "throw" and "throws" before. Go into the API guide and bring up the class `String`. Now scroll down to the method summary and click on the link for the familiar method `charAt()`. You will see this notation.

Throws:

> `IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of the string.

Let us now look up this `IndexOutOfBoundsException`. The family tree reveals that this class extends the `RuntimeException` class. The purpose of this exception is to terminate the execution of the program since you are in an error state. We can, however, run code in response to the occurrence of an exception, so our program does not crash.

5 Throwing an Exception

You can throw exceptions when something is about to go wrong. For example in our BigFraction class, we threw an exception if a denominator of zero is passed. Here it is.

```
public BigFraction(BigInteger _num, BigInteger _denom)
{
    if(denom.equals(BigInteger.ZERO)
    {
        throw new IllegalArgumentException();
    }
    (rest of constructor)
}
```

It is best, if at all possible to use a standard library exception. You can create your own exceptions by inheriting from any class in the `Exception` subtree. If you inherit from `RuntimeException`, the caller does not have to handle the exception as shown in the next section.

6 Checked and Run-Time Exceptions

There are two types of exceptions that exist: `RuntimeExceptions` and all others, which are called *checked exceptions*. Generally a run-time exception is caused by *programmer* error. Programmers should know better, and it is probably good for them to have their programs using your class die with an exploding heart as a reward for writing rotten code. Such programmers can read the stack trace and lick their wounds. How do you know if an exception is a `RuntimeException`? Just look up its family tree and see if it is a descendant of `RuntimeException`. So far in our study of Java, we have only seen runtime exceptions.

Checked exceptions, on the other hand, are usually caused by situations beyond programmer, or even end-user control. Suppose a user tries to get a program to open a file that does not exist, or a file for which he lacks appropriate permissions. Another similar situation is that of attempting to create a *socket*, or a connection to another computer. The host computer may not allow such connections, it could be down, or it could even be nonexistent. These situations are not necessarily the user's or programmer's fault.

Checked exceptions must be *handled*; this process entails creating code to tell your program what to do in the face of these exceptions being thrown. It is entirely optional to handle a runtime exception.

Sometimes a runtime exception will be caused by user error; in these cases it is appropriate to use exception handling to fix the problem. For example if a user is supposed to enter a number into a `TextField` the hapless fool enters a string that is not numeric, your program might try to use `Integer.parseInt` to convert it into an integer. Here we see a problem created by an end-user. This user should be protected and this error should be handled gracefully so that (bumbling) user can go about his business. You always want to protect the end-user from exceptions if it is at all feasible or reasonable. Remember: *Never reward a paying customer with death.* It's bad for business.

6.1 Catching It

Java provides a parallel track of execution for handling exceptions gracefully. Suppose you are writing a program that displays a color in response to a hex code entered by a (very dumb) end-user of Your Shining Program. The user enters something like `ffgg00`; this is a situation that you, the programmer do not control. The Java exception mechanism would allow you to cleanly punt

and reset your color calculator to some state, such as white, and display the appropriate hex code, `0xFFFFFF`.

Some resourceful hackish readers might think, "Here is a new and useful way to get unwedged from a bad situation." This is a mistake. Only use exception handling for error situations beyond programmer control. Do not use them for the ordinary execution of your programs. Unwinding the call stack for an exception is a costly operation.

7 A Simple Case Study

Let us write a simple color calculator. Our application is to have three graphical elements. It will have a color panel to display the color sample which will occupy most of the frame. On the top of the frame we will place a button and a textfield. The user types into the textfield, hits enter or hits the button, and the color is shown.

Let us start by blocking out the key elements of our application. We need state variables for a border pane, a color, and a canvas. We supply an `init` method to initilaize things and put them in their places.

We have an as-yet empty `refresh` method to tell the canvas to repaint itself.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
public class SimpleColorCalc extends Application
{
    private BorderPane bp;
    private Color displayed;
    private Canvas canvas;
    @Override
    public void init()
    {
        bp  = new BorderPane();
        displayed = Color.WHITE;
        canvas = new Canvas(400,300);
        refresh();
        bp.setCenter(canvas);
    }
    private void refresh()
    {
    }
```

```
    public static void main(String[] args)
    {
        launch(args);
    }
    @Override
    public void start(Stage primary)
    {
    }
}
```

The `refresh` method is simple to write. It just needs to color the canvas the color `displayed`.

```
private void refresh()
{
    GraphicsContext g = canvas.getGraphicsContext2D();
    g.setFill(displayed);
    g.fillRect(0, 0, canvas.getWidth(), canvas.getHeight());
}
```

Also, you need to add this import.

```
import javafx.scene.canvas.GraphicsContext;
```

Compile. If you run it, nothing will show since we have no scene and the stage has not been told to show. You can add this code to see it to the empty `start` method.

```
primary.setScene(new Scene(bp));
primary.show();
```

Now let us get some GUI elements into place. Let us add some items to `start`. We will create a text field and a button and place them in a `FlowPane`, along with a label. We then will pop the `FlowPane` into the top of the `BorderPane`.

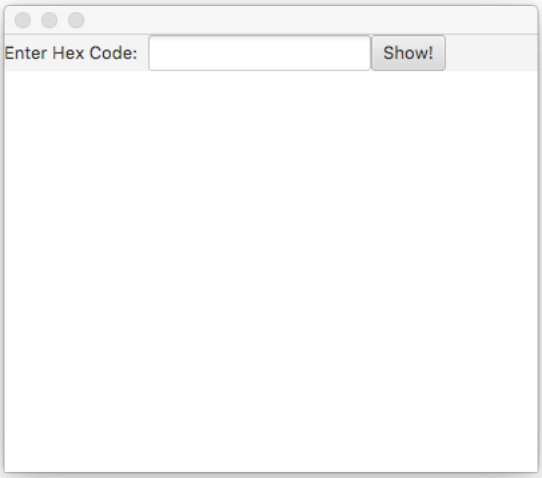
```
@Override
public void start(Stage primary)
{
    TextField tf = new TextField();
    Button showButton = new Button("Show!");
    FlowPane fp = new FlowPane();
    bp.setTop(fp);
    fp.getChildren().addAll(new Label("Enter Hex Code:  "), tf, showButton);
```

```
    primary.setScene(new Scene(bp));
    primary.show();
}
```

Add these imports.

```
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.FlowPane;
```

Here is the result of running the program.



1. Get the text from the `JTextField`. It comes in as a string.
2. Turn it into a hex code.
3. Get the color for the hex code
4. Have the color panel paint itself that color.

We will now create event handling code for the button.

```
showButton.setOnAction( e ->
{
    String s = tf.getText();
    if(s.startsWith("0x"))
```

```
    {
        s = s.substring(2);  //cut off lead 0x
    }
    int code = Integer.parseInt(s, 16);
    displayed = Color.web(String.format("#%06x", code), 1.0);
    refresh();
});
```

There is still an ugly reality: the inept end-user might type in some unparseable gobbledegook and spoil our program's irenic existence. The user just might do something stupid like enter `cowpie` in the `JTextField`. We now turn to exception handling to prevent this sort of nonsense. Do so at this point and watch the ugliness spew.

If you refere to the API page for `Integer`, You can also see that `parseInt` throws a `NumberFormatException`. Go to the API page for this exception; you can do so by clicking on the link shown.

This is a runtime exception, as you can see by looking up the family tree. However, it is triggered by an end-user's blunder, so we shall deal with it gracefully. We will just display `ERROR` in the text field and bail.

Imagine you are about to get into a tub full of water. Do you just plunge in? This is not the recommended course of action if you desire continued and comfortable existence. You first dip a finger or toe in the water. If it's too hot or too cold, you throw a `BadTubwaterTemperatureException`. You can recover from such an error. If the tub is too cold, add hot water until the desired temperature is reached. If the tub is too hot, let it cool or add cold water until the water is at a suitable temperature.

Java provides a mechanism called the the `try-catch` sequence to handle the exception. We now insert this and explain it.

```
showButton.setOnAction( e ->
{
    String s = tf.getText().toLowerCase();
    if(s.startsWith("0x"))
    {
        s = s.substring(2);  //cut off lead 0x
    }
    try
    {
        int code = Integer.parseInt(s, 16);
        displayed = Color.web(String.format("#%06x", code), 1.0);
        refresh();
    }
    catch(NumberFormatException ex)
    {
```

```
        tf.setText("ERROR");
        return;
    }
});
```

You place the "dangerous code" you are using inside of the `try` block. In this case, the danger is generated by a number format exception triggered by the abuse of our innocent program. If the user enters a legal hex code, the `catch` block is ignored; if not, the `catch` block executes. This precludes the occurrence of an exploding heart caused by an end–user abusing `Integer.parseInt()`. Once we have circumnavigated the danger, we can go about our business of obtaining a color and coloring the panel. Notice that all of the code using the integer variable `code` is placed in the `try` block. This is because an exception will cause the `try` block to abort immediately, and the variable you wish to use will never be created during an error state.

Can we make the same event-handling code work if we hit the enter key instead of clicking on the button? Yes. We could make a call to `setOnAction` for the textfield and insert the same event-handling code as we did for the button. This is a violaton of the 11th commandment. Instead, let us do this.

Begin by adding these imports.

```
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
```

Now what we do is make a single `EventHandler<ActionEvent>` object using a lambda and we attach it to the button and text field.

```
@Override
public void start(Stage primary)
{
    TextField tf = new TextField();
    Button showButton = new Button("Show!");
    EventHandler<ActionEvent> colorReact =  e ->
    {
        String s = tf.getText().toLowerCase();
        if(s.startsWith("0x"))
        {
            s = s.substring(2);  //cut off lead 0x
        }
        try
        {
            int code = Integer.parseInt(s, 16);
```

```
            displayed = Color.web(String.format("#%06x", code), 1.0);
            refresh();
        }
        catch(NumberFormatException ex)
        {
            tf.setText("ERROR");
            return;
        }
    };
    showButton.setOnAction(colorReact);
    tf.setOnAction(colorReact);
    FlowPane fp = new FlowPane();
    bp.setTop(fp);
    fp.getChildren().addAll(new Label("Enter Hex Code:  "), tf, showButton);
    primary.setScene(new Scene(bp));
    primary.show();
    }
}
```

All Code Shown

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.TextField;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.FlowPane;
import javafx.event.EventHandler;
import javafx.event.ActionEvent;
public class SimpleColorCalc extends Application
{
    private BorderPane bp;
    private Color displayed;
    private Canvas canvas;
    @Override
    public void init()
    {
        bp  = new BorderPane();
        displayed = Color.WHITE;
        canvas = new Canvas(400,300);
```

```
        refresh();
        bp.setCenter(canvas);
    }
    private void refresh()
    {
        GraphicsContext g = canvas.getGraphicsContext2D();
        g.setFill(displayed);
        g.fillRect(0, 0, canvas.getWidth(), canvas.getHeight());
    }
    public static void main(String[] args)
    {
        launch(args);
    }
@Override
public void start(Stage primary)
{
    TextField tf = new TextField();
    Button showButton = new Button("Show!");
    EventHandler<ActionEvent> colorReact =  e ->
    {
        String s = tf.getText().toLowerCase();
        if(s.startsWith("0x"))
        {
            s = s.substring(2);  //cut off lead 0x
        }
        try
        {
            int code = Integer.parseInt(s, 16);
            displayed = Color.web(String.format("#%06x", code), 1.0);
            refresh();
        }
        catch(NumberFormatException ex)
        {
            tf.setText("ERROR");
            return;
        }
    };
    showButton.setOnAction(colorReact);
    tf.setOnAction(colorReact);
    FlowPane fp = new FlowPane();
    bp.setTop(fp);
    fp.getChildren().addAll(new Label("Enter Hex Code:  "), tf, showButton);
    primary.setScene(new Scene(bp));
    primary.show();
    }
}
```

8 And Now Back to NitPad

Let us begin to think about the user experience. In the beginning, we will see a window with menus and an empty `TabPane`. The user will select the `New` menu item from the `File` menu and a tab will appear. The tab will be an instance of the `Tab` class.

8.1 Paying the `Tab`

The tab will have a little label at the top; the text in this label can be set with `setText` method for tabs. Look at the constructor summary. There are three constructors for `Tab`.

- `Tab()` This creates an empty tab with no text label.
- `Tab(String text)` This creates and empty tab with `text` as its tab label.
- `Tab(String text, Node content)` This creates a tab with the specified content and with tab label `text`.

We cannot enter text directly into a tab; we need a control that accepts text. For this purpose, we will use the `TextArea`. So, we see right off we need for our tabs to contain text areas. We will therefore avail ourselves of inheritance. We will create a new class called `TextTab`, which will hold text. Each text tab will need to be aware of the file it is now working on. Take notice that nearly all of the methods in this class are marked `final`, so we cannot override them.

So, let us begin to create the class `TextTab`. Note that we are *not* extending `Application`.

```
import javafx.scene.control.Tab;
public class TextTab extends Tab
{
}
```

Now it's time for a little state. We put in our text area and make the file that the tab is seeing state as well. Since we will never change the text area, we will mark it `final`. If there is no file being seen, we make the `file` be `null`.

```
import javafx.scene.control.Tab;
import java.io.File;
public class TextTab extends Tab
{
    private final TextArea ta;
    private File file;
```

```
    public TextTab()
    {
        ta = new TextArea();
        file = null;
    }
}
```

Let us now allow for the file to be changed or seen, and for the text to be hoovered out of the text area.

```
import javafx.scene.control.Tab;
import javafx.scene.control.TextArea;
import java.io.File;
public class TextTab extends Tab
{
    private TextArea ta;
    private File file;
    public TextTab()
    {
        ta = new TextArea();
        file = null;
    }
    public File getFile()
    {
        return file;
    }
    public void setFile(File newFile)
    {
        file = newFile;
    }
    public String hooverText()
    {
        return ta.getText();
    }
    public TextArea getTextArea()
    {
        return ta;
    }
}
```

We shall assign each tab we create a unique ID. This will be done by cribbing from the program `Minter.java` shown earlier. We will make the tab's `ID` be public and final. It is safe to do so, since `int` is a primitive type and, in this case, `ID` is constant. Notice that we hide the `IDWell` and provide no means for client programmer access.

```
import javafx.scene.control.Tab;
import javafx.scene.control.TextArea;
import java.io.File;
public class TextTab extends Tab
{
    private static int IDWell;
    static
    {
        IDWell = 0;
    }
    public final int ID;
    private TextArea ta;
    private File file;
    public TextTab()
    {
        ta = new TextArea();
        file = null;
        IDWell++;
        ID = IDWell;
    }
    public File getFile()
    {
        return file;
    }
    public void setFile(File newFile)
    {
        file = newFile;
    }
    public String hooverText()
    {
        return ta.getText();
    }
    public TextArea getTextArea()
    {
        return ta;
    }
}
```

Now add these lines to the constructor, so a tab now has a text area in it and so its ID is displayed.

```
        setText(String.format("Tab %s", ID));
        setContent(ta);
```

8.2 Getting Tabs into the Tab Pane

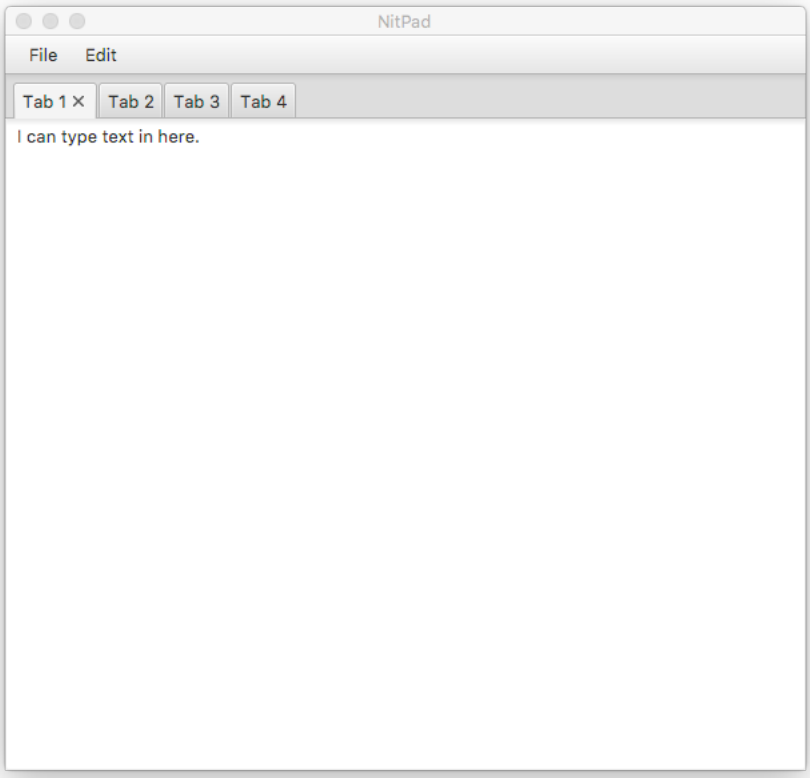
Let us now create the `New` menu and have it create new tabs. Here is the current status of our file menu.

```
private void createFileMenu()
{
    Menu fileMenu = new Menu("File");
    mbar.getMenus().addAll(fileMenu);
}
```

Let us create a `New` menu.

```
private void createFileMenu()
{
    Menu fileMenu = new Menu("File");
    mbar.getMenus().addAll(fileMenu);
    MenuItem newItem = new MenuItem("New");
    fileMenu.getItems().addAll(newItem);
    newItem.setOnAction( e ->
    {
        tp.getTabs().add(new TextTab());
    });
}
```

Now select the `New` menu item. Notice how sequentially numbered tabs appear. Also notice you can type right into the tabs. Notice that `Tab 1` stays as the active tab; you must click on the others to gain access to them.



Let us now figure out how to make the new tab be the active tab.

To accomplish this, we call `tp.getSelectionModel.select()` on a tab. We now modify our `newItem`'s event handling code as follows. Note that we need to name the tab in a local variable to work with it. We maintain a state variable to keep track of the tab that is currently selected.

```
newItem.setOnAction( e ->
{
    Tab t = new TextTab();
    activeTab = t;
    tp.getTabs().add(t);
    tp.getSelectionModel().select(t);
});
```

Now when you run the program, the new tab will be the one selected.

Programming Exercises

1. Can you figure out how to add all new tab on the left instead of the right?
2. Can you make the new tab be added just to the right of the currently active tab?

9 Opening a File

We next create a menu item to open a file. When the file is opened, it should be displayed in a new tab. We will create a status bar on the bottom of our app and update the filename appearing in the status bar. If there is no currently open file, we should simply display `"New File"` in the status bar. If the file is saved, we will put a `*` at the end of the file's path.

So who keeps track of this? Each tab has a state variable for the file it now has open, so we will make this the job of the tab. We will add this state variables to `TextTab.java`.

```
private isSaved;
```

When a new tab is created, we will mark it as saved so we do not end up annoying the user with requests to save unused tabs. However, as soon as the tab's text area gets changed, we need to mark `isSaved` as `false`. We modify the `TextTab` constructor as follows.

```
public TextTab()
{
    ta = new TextArea();
    //adding a change listener to the text area
    ta.textProperty().addListener( (obs, oldValue, newValue) ->
    {
        isSaved = false;
    });
    file = null;
    IDWell++;
    ID = IDWell;
    setText(String.format("Tab %s", ID));
    setContent(ta);
}
```

Now let's add the status bar. We can make it an `HBox` and place a label inside of it. The label will need to be a state variable, so we can set its text from within the NitPad class. Our NitPad class now looks like this.

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuItem;
import javafx.scene.control.MenuBar;
import javafx.scene.control.TabPane;
import javafx.scene.control.Tab;
import javafx.scene.layout.BorderPane;
import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
public class NitPad extends Application
{
    private TabPane tp;
    private BorderPane bp;
    private MenuBar mbar;
    private Tab activeTab;
    private Label label;
    public static void main(String[] args)
    {
        launch(args);
    }
    @Override
    public void init()
    {
        tp = new TabPane();
        bp = new BorderPane();
        mbar = new MenuBar();
        label = new Label("New File");
    }
    @Override
    public void start(Stage primary)
    {
        primary.setTitle("NitPad");
        bp.setTop(mbar);
        bp.setCenter(tp);
        createFileMenu();
        createEditMenu();
        HBox statusBar = new HBox();
        statusBar.getChildren().add(label);
        bp.setBottom(statusBar);
        primary.setScene(new Scene(bp, 600,550));
        primary.show();
    }
    private void createFileMenu()
    {
```

```
        Menu fileMenu = new Menu("File");
        mbar.getMenus().addAll(fileMenu);
        MenuItem newItem = new MenuItem("New");
        fileMenu.getItems().addAll(newItem);
        newItem.setOnAction( e ->
        {
            Tab t = new TextTab();
            activeTab = t;
            tp.getTabs().add(t);
            tp.getSelectionModel().select(t);
            label.setText("New File");
        });
    }
    private void createEditMenu()
    {
        Menu editMenu = new Menu("Edit");
        mbar.getMenus().addAll(editMenu);
    }
}
```

Now we are ready to open a file. Here are some things we need to do

1. Add a menu item `Open...`
2. Now add event handling code that does the following:
   (a) pops up a `FileChooser`
   (b) selects a file
   (c) aspirates the contents of the file into a `StringBuffer`
   (d) closes the file
   (e) places the text in a new tab and makes that tab active.

Go ahead and add the menu item and see that it appears in the `File` menu. You can also do these imports, as we will need them soon.

```
import java.io.File;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.io.FileNotFoundException;
```

Also, place the import for `java.io.File` into your `NitPad` class. Now we write the method to take a tab's file and place it in

```
//returns true if text successfully displayed.
public boolean displayText()
{
    if(file == null)
        return false;   ///bail if file is null
    try
    {
        BufferedReader br = new BufferedReader(new FileReader(file));
        String line;
        StringBuffer sb = new StringBuffer();
        while( (line = br.readLine()) != null)
        {
            sb.append(line + "\n");
        }
        ta.setText(sb.toString());
        br.close();
        return true;
    }
    catch(FileNotFoundException ex)
    {
        ta.setText(String.format("File %s not found", file.getAbsolutePath()));
        return false;
    }
    catch(IOException ex)
    {
        ta.setText(String.format("File %s triggered IOException.", file.getAbsolutePath(
        return false;
    }
}
```

Finally, here is the event-handling code which you should place in `NitPad`'s `createFileMenu` method.

```
openItem.setOnAction( e ->
{
    FileChooser fc = new FileChooser();
    fc.setTitle("Open File");
    File file = fc.showOpenDialog(new Stage());
    if(file == null)
        return;  //user balked; do nothing
    TextTab t = new TextTab();
    activeTab = t;
    t.setFile(file);
    tp.getTabs().add(t);
    tp.getSelectionModel().select(t);
```

```
        t.displayText();
        label.setText(t.getFile().getAbsolutePath());
    });
```

You should now be able to select a file and display its contents to a new tab. This will also display that file's path in the status bar.

10 Saving a File from a Tab and Managing the Active Tab

Let us begin by adding three new menu items, `Save`, `Save As...`, and `Quit`. We add these lines of code to `createFileMenu`

```
        MenuItem saveItem = new MenuItem("Save");
        MenuItem saveAsItem = new MenuItem("Save As...");
        MenuItem quitItem = new MenuItem("Quit");
```

Now add the menu items to the `File` menu.

```
    fileMenu.getItems().addAll(newItem, openItem, saveItem,
        saveAsItem, quitItem);
```

One item we need to pay attention to is the active tab. When we make a new tab by selecting `New` or `Open...` we should make that tab the active tab. It is convenient to place the code for adjusting the status bar in its own place. Hence, we do this.

```
    private void updateStatusBar()
    {
        //if no tabs are open, do nothing
        if(activeTab == null)
        {
            return;
        }
        if(activeTab.getFile() == null)
        {
            label.setText("New File");
            return;
        }
        label.setText(activeTab.getFile().getAbsolutePath());
    }
```

Let us look at how the management of the active tab is done by each of the menu items. The `New` item. It simply assigns the active tab to be the tab that just got created and it selects it.

```
newItem.setOnAction( e ->
{
    TextTab t = new TextTab();
    activeTab = t;
    tp.getTabs().add(t);
    tp.getSelectionModel().select(t);
    t.setFile(null);
        t.setSaved(true);
    updateStatusBar();
});
```

The `Open...` item manages the active tab the same way as `New`.

```
openItem.setOnAction( e ->
{
    FileChooser fc = new FileChooser();
    fc.setTitle("Open File");
    File file = fc.showOpenDialog(new Stage());
    if(file == null)
        return;  //user balked; do nothing
    TextTab t = new TextTab();
    activeTab = t;
    t.setFile(file);
    tp.getTabs().add(t);
    tp.getSelectionModel().select(t);
    t.displayText();
    updateStatusBar();
});
```

When saving, we save from the active tab. If there is not active tab, we will do nothing. If the active tab has no file, we will ask the user to choose a file. Once the file is saved, we mark the tab as being saved.

```
saveItem.setOnAction( e ->
{
    File file = null;
    if(activeTab == null)
        return;   //pane is empty; nothing to save
    if(activeTab.getFile() == null)
    {
        FileChooser fc = new FileChooser();
```

```
        fc.setTitle("Save File");
        file = fc.showSaveDialog(new Stage());
        if(file == null)
            return;  //user balked; do nothing
    }
    activeTab.setFile(file);
    activeTab.saveDisplayedText();
        activeTab.setSaved(true);
    label.setText(activeTab.getFile().getAbsolutePath());
});
```

Now we write the `Save As` event handling. Note that we save a copy and we do not change the current file.

```
saveAsItem.setOnAction( e ->
{
    File file = null;
    if(activeTab == null)
        return;   //pane is empty; nothing to save
    FileChooser fc = new FileChooser();
    fc.setTitle("Save File");
    file = fc.showSaveDialog(new Stage());
    if(file == null)
       return;  //user balked; do nothing
    activeTab.setFile(file);
    activeTab.saveDisplayedText();
    activeTab.setSaved(true);
});
```

Finally, let us write the quit event handling code. This must walk through each unsaved tab and offer the user a chance to save the file there.

```
quitItem.setOnAction( e ->
{
    for(Tab t: tp.getTabs())
    {
        TextTab tt = (TextTab) t;
        if(!tt.tellIfSaved())
        {
            //offer to choose file
            FileChooser fc = new FileChooser();
            fc.setTitle(String.format("Save %s? Cancel to skip saving.", tt.getText()));
            File destination = fc.showSaveDialog(new Stage());
            if(destination == null)
                return;  //user balked; do not save
```

```
            tt.setFile(destination);
            tt.saveDisplayedText();
        }
    }
    Platform.exit();
});
```

We now have a completed menu for `File`.

11 The Edit Menu

We will build in the basic features. Here are the menu items we will create

- `Copy` copies selected text to system clipboard
- `Cut` cuts selected text to system clipboard
- `Delete` deletes selected text and leaves system clipboard unchanged
- `Paste` pastes contents to clipboard to place where cursor is
- `Select All` selects the entire document