CITS1001 week 6
Libraries

Arran Stewart

April 12, 2018

Announcements

- Project 1 available
- mid-semester test
- self-assessment

Outline

- Using library classes to implement some more advanced functionality
  - Using library classes
  - Reading documentation
- Reading: Chapter 6 of Objects First with Java - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling

The Java class library

- Thousands of classes.
- Tens of thousands of methods.
- Many useful classes that make life much easier.
- Library classes are often inter-related.
- Arranged into packages.

Working with the library

- A competent Java programmer must be able to work with the libraries.
- You should:
  - know some important classes by name;
  - know how to find out about other classes.
- Remember:
  - we only need to know the *interface*, not the implementation.

Example: an interactive text system

Main loop structure

```
boolean finished = false;
while(!finished) {
    // do something...
    if( /* exit condition .. */ ) {
        finished = true;
    } else {
        // do something more
    }
}
```

- This is a common iteration pattern.

Main loop body

```
String input = reader.getInput();
//...
String response = responder.generateResponse();
System.out.println(response);
```

The exit condition

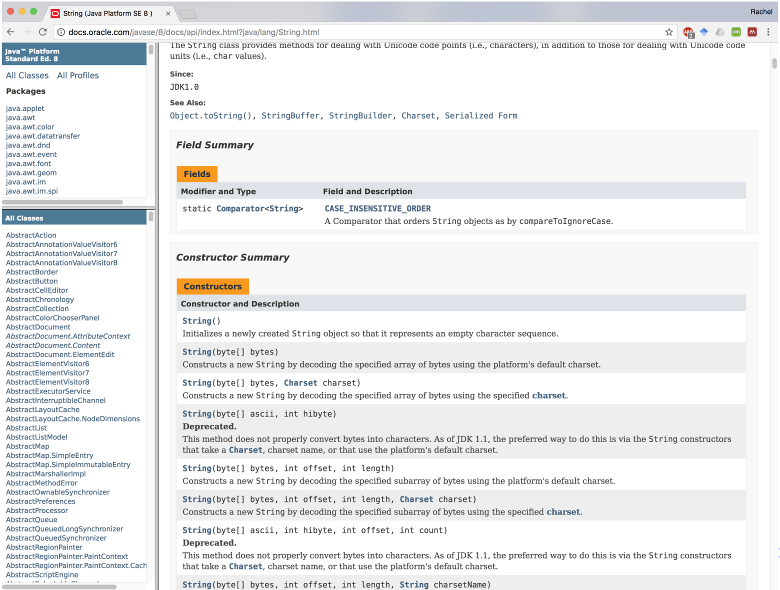
```
String input = reader.getInput();
if(input.startsWith("bye")) {
    finished = true;
}
```

- Where does ‘startsWith’ come from?
- What is it? What does it do?
- How can we find out?

Reading class documentation

- Documentation of the Java libraries is in HTML format
  - readable in a web browser
- Provides an *API* (Application Programmers' Interface) for the classes
  - i.e. an interface description for all library classes
- Address: http://docs.oracle.com/javase/8/docs/api
  (or just Google "oracle java API")

String class



String (Java Platform SE 8)
Rachel
docs.oracle.com/javase/8/docs/api/index.html?java/lang/String.html

Java™ Platform Standard Ed. 8

All Classes All Profiles

Packages

java.applet
java.awt
java.awt.color
java.awt.datatransfer
java.awt.dnd
java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi

All Classes

AbstractAction
AbstractAnnotationValueVisitor6
AbstractAnnotationValueVisitor7
AbstractAnnotationValueVisitor8
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractChronology
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.AttributeContext
AbstractDocument.Content
AbstractDocument.ElementEdit
AbstractElementVisitor6
AbstractElementVisitor7
AbstractElementVisitor8
AbstractExecutorService
AbstractInterruptibleChannel
AbstractLayoutCache
AbstractLayoutCache.NodeDimensions
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutableEntry
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchronizer
AbstractPreferences
AbstractProcessor
AbstractQueue
AbstractQueuedLongSynchronizer
AbstractQueuedSynchronizer
AbstractRegionPainter
AbstractRegionPainter.PaintContext
AbstractRegionPainter.PaintContext.Cach
AbstractScriptEngine

The String class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).

Since:

JDK1.0

See Also:

Object.toString(), StringBuffer, StringBuilder, Charset, Serialized Form

Field Summary

Fields

Modifier and Type	Field and Description
static Comparator<String>	CASE_INSENSITIVE_ORDER A Comparator that orders String objects as by compareToIgnoreCase.

Constructor Summary

Constructors

Constructor and Description
String() Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, Charset charset) Constructs a new String by decoding the specified array of bytes using the specified **charset**.
String(byte[] ascii, int hibyte) **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.
String(byte[] bytes, int offset, int length) Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.
String(byte[] bytes, int offset, int length, Charset charset) Constructs a new String by decoding the specified subarray of bytes using the specified **charset**.
String(byte[] ascii, int hibyte, int offset, int count) **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.
String(byte[] bytes, int offset, int length, String charsetName)

String class

- (Documentation is at http://docs.oracle.com/javase/8/docs/api/index.html?java/lang/String.html)

Interface vs implementation

The documentation includes:

- the name of the class;
- a general description of the class;
- a list of (public) constructors and methods
- return values and parameters for constructors and methods
- a description of the purpose of each constructor and method

All this comprises the *interface* of the class

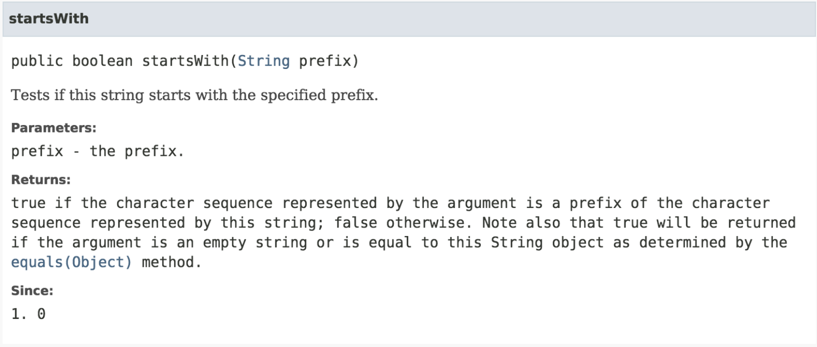
Interface vs implementation

The documentation does not include:

- private fields (most fields are private)
- private methods
- the bodies (source code) of methods

These (hidden) details comprise the *implementation* of the class

Documentation for startsWith



startsWith

```
public boolean startsWith(String prefix)
```

Tests if this string starts with the specified prefix.

Parameters:

`prefix - the prefix.`

Returns:

`true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise. Note also that true will be returned if the argument is an empty string or is equal to this String object as determined by the equals(Object) method.`

Since:

`1. 0`

Methods from String

- `contains`
- `endsWith`
- `indexOf`
- `substring`
- `toUpperCase`
- `trim`

On the topic of Strings:

- note that Strings are *immutable* –
    - once it is created, a String object cannot be changed.
- The String class has a number of methods that *appear* to modify strings –
    - Since Strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.
- See https://docs.oracle.com/javase/tutorial/java/data/strings.html
- (Do you know if any classes we have seen are immutable?)

Using library classes

- *Classes* are organized into *packages*.
- To use a class from the library, it must be *imported* using an **`import`** statement (except classes from the **`java.lang`** package).
- Once imported, a class can then be used like classes from the current project.

Packages and import

Using an import statement ...

- ... we can import *single* classes:

```
import java.util.ArrayList;
```

- ... and whole packages of classes:

```
import java.util.*;
```

- Importation does not involve source code insertion.

Example: using Random

- The library class Random can be used to generate random numbers

```
import java.util.Random;
// ...
Random rand = new Random();
// ...
int num = rand.nextInt();
int value = 1 + rand.nextInt(100);
int index = rand.nextInt(list.size());
```

Examples cont'd

- How could we fill an `ArrayList` with (say) 10 random integers between 0 and 25 (inclusive)?

Examples cont'd

- How could we fill an `ArrayList` with (say) 10 random integers between 0 and 25 (inclusive)?

-
  ```
  Random randomGenerator = new Random();
  ArrayList nums = new ArrayList<Integer>();
  for (int i = 0; i < 10; i++) {
    int num = rand.nextInt(26);
    nums.add(num)
  }
  ```

Examples cont'd

- Suppose we have an `ArrayList` containing Strings. How can we print a randomly selected String from it?

Examples cont'd

- Suppose we have an `ArrayList` containing Strings. How can we print a randomly selected String from it?
-
  ```
  ArrayList<String> myStrings = //..
  // ..
  Random randomGenerator = new Random();
  int idx = rand.nextInt(myStrings.size());
  System.out.println( myStrings.get(idx) );
  ```

Parameterized classes

- For some classes, the documentation includes provision for a *type parameter*:
  - `ArrayList<E>`
- These type names reappear in the parameters and return types:
  - `E get(int index)`
  - `boolean add(E e)`

Parameterized classes

- The types in the documentation are placeholders for the types we use in practice
- Given the generic ArrayList methods ...
  - `E get(int index)`
  - `boolean add(E e)`

  ... once we declare that something is an **`ArrayList<TicketMachine>`**, it will end up having the following methods:

  - `TicketMachine get(int index)`
  - `boolean add(TicketMachine e)`

Documentation and more advanced collections

Main concepts to be covered

We look at using library classes to implement some more advanced functionality.

- Further library classes
  - `Set`
  - `Map`
- Writing documentation
  - `javadoc`

Using sets

```
import java.util.HashSet;
//...
HashSet<String> mySet = new HashSet<String>();
mySet.add("one");
mySet.add("two");
mySet.add("three");
for(String element : mySet) {
    // do something with element
}
```

- Hopefully seems quite similar to an `ArrayList`

Set example – words in a String

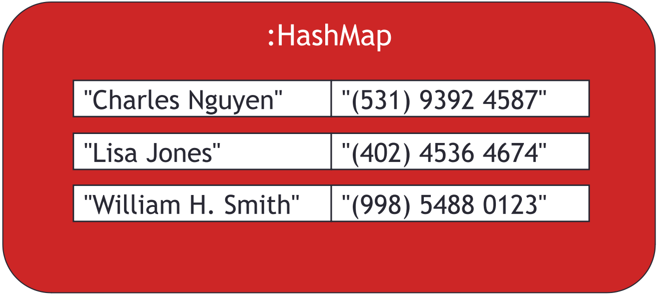
```
public HashSet<String> getInput() {
    System.out.print("> ");
    String inputLine =
        reader.nextLine().trim().toLowerCase();
    String[] wordArray = inputLine.split(" ");
    HashSet<String> words = new HashSet<String>();
    for(String word : wordArray) {
        words.add(word);
    }
    return words;
}
```

Maps

- Maps are collections that contain *pairs* of values.
- Pairs consist of a *key* and a *value*.
- Lookup works by supplying a key, and retrieving a value.
- Example: a telephone book.

Using maps

- A map with strings as keys and values



:HashMap

"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

Using maps

```
HashMap <String, String> phoneBook =
          new HashMap<String, String>();
phoneBook.put("Charles Nguyen", "(531) 9392 4587");
phoneBook.put("Lisa Jones", "(402) 4536 4674");
phoneBook.put("William H. Smith", "(998) 5488 0123");

String phoneNumber = phoneBook.get("Lisa Jones");
System.out.println(phoneNumber);
```

- Aside: why "*hash*map"?

List, Map and Set

- Alternative ways to group objects.
- Varying implementations available:
  - Lists: `ArrayList`, `LinkedList`
  - Sets: `HashSet`, `TreeSet`

Writing class documentation

- Your own classes should be documented the same way library classes are.
- Other people should be able to use your class without reading the implementation.
- i.e. Make your class a potential ‘library class’

Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number
- the authors' names
- documentation for each constructor and each method

Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type
- the parameter names and types
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned

Elements of documentation

purpose and function:

- what does the caller need to ensure when calling the method?
- if exceptions (we'll cover these) are thrown, what sorts can they be?
- if *side effects* happen (i.e. something other than returning a value), what are they?
    - e.g. printing to the screen; writing to a database or file; changing the state of an object
    - (often `void` methods, but not always)

javadoc

Class comment:

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 1.0 (2011.07.31)
 */
```

javadoc

Method comment:

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param prompt A prompt to print to screen.
 * @return A set of Strings, where each String is
 *    one of the words typed by the user
 */
public HashSet<String> getInput(String prompt)
{
   // ...
}
```

Public vs private

- Public elements are accessible to objects of other classes:
    - Fields, constructors and methods
- Fields should not (usually) be public.
- Private elements are accessible only to objects of the same class.
- Only methods that are intended for other classes should be public.

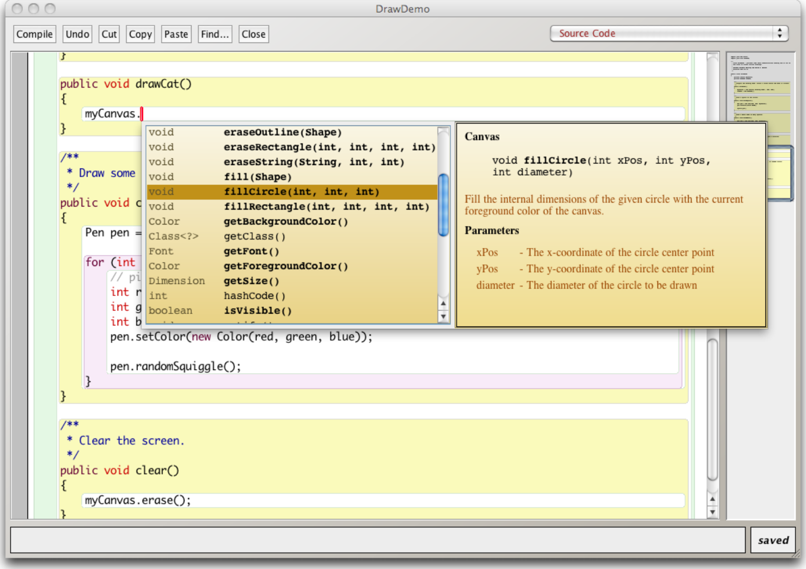
Information hiding

- Data belonging to one object is hidden from other objects.
- Know what an object can do, not how it does it.
- Information hiding increases the level of independence.
- Independence of modules is important for large systems and maintenance.

Aside: code completion in BlueJ

- The BlueJ editor supports lookup of methods.
- Use Ctrl-space after a method-call dot to bring up a list of available methods.
- Use Return to select a highlighted method.

Code completion in BlueJ



```
}

public void drawCat()
{
    myCanvas.
}

/**
 * Draw some
 */
public void c
{
    Pen pen =

    for (int
        // pi
        int r
        int g
        int b
        pen.setColor(new Color(red, green, blue));

        pen.randomSquiggle();
    }
}

/**
 * Clear the screen.
 */
public void clear()
{
    myCanvas.erase();
}
```

```
void        eraseOutline(Shape)
void        eraseRectangle(int, int, int, int)
void        eraseString(String, int, int)
void        fill(Shape)
void        fillCircle(int, int, int)
void        fillRectangle(int, int, int, int)
Color       getBackgroundColor()
Class<?>    getClass()
Font        getFont()
Color       getForegroundColor()
Dimension   getSize()
int         hashCode()
boolean     isVisible()
```

Canvas

```
void fillCircle(int xPos, int yPos,
int diameter)
```

Fill the internal dimensions of the given circle with the current foreground color of the canvas.

Parameters

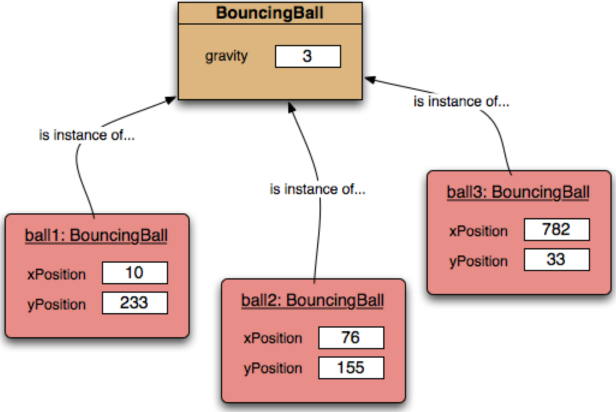
xPos - The x-coordinate of the circle center point
yPos - The y-coordinate of the circle center point
diameter - The diameter of the circle to be drawn

Class and constant variables

Class variables

- A class variable is shared between all instances of the class.
- In fact, it belongs to the class and exists independent of any instances.
- Designated by the `static` keyword.
- Public static variables are accessed via the class name; e.g.:
  - `Thermometer.boilingPoint`

Class variables



Constants

- A variable, once set, can have its value fixed.
- Designated by the `final` keyword.
  - `final int max = list.size();`
- Final fields must be set in their declaration or the constructor.
- Combining `static` and `final` is common.

Class constants

- `static`: class variable
- `final`: constant

```
private static final int gravity = 3;
```

- Public visibility is less of an issue with `final` fields.
- Upper-case names often used for class constants:

```
public static final int BOILING_POINT = 100;
```

Using the class Math

- Whenever you need a mathematical function, it will (probably) be in the class **Math**
- **java.lang.Math** (can be referred to just as **Math**)
- For example, Java does not have a built-in power operator, but it is available in **Math**

```
public static double circleArea(double radius) {
  double area = 3.14159 * Math.pow(radius,2);
  return area;
}
```

Math.random()

```
public static double random()
```

Returns a double `x` such that `0.0 <= x < 1.0`

(Try it in the BlueJ Code Pad)

Example:

```
boolean isheads = Math.random() < 0.5;
```

Math Constants

- Class variables are often used to provide access to constants – values that are frequently used but not changed
- Constants can be numerical values
    - `Math.PI`
    - `Math.E`

```
public static double circleArea(double radius) {
  return Math.PI * Math.pow(radius,2);
}
```

Utility Classes

- A class like **`Math`** that contains only static methods is sometimes called a *utility* class, because it just provides “utility” methods for use in other classes
- There is no point in ever creating an *object* of the class **`Math`** because it can never do anything that the existing methods cannot do
- (In fact it has been made impossible to create an object of the class **`Math`**
    - this is done by giving a dummy constructor, but making it private)

Constant Objects: Colours

- The class `java.awt.Color` makes available a number of “pre-constructed” objects

```
Color.RED
Color.BLUE
...
Color.BLACK
```

- You can use these colours without having to construct them from scratch
- See http://teaching.csse.uwa.edu.au/units/CITS1001/colorinfo.html for more information about colors in Java

Practice

- Write constant declarations for the following:
  - A public variable to measure tolerance, with the value 0.001
  - A private variable to indicate a pass mark, with integer value of 40
  - A public character variable that is used to indicate that the help command is ‘h’.
- What constant names are defined in the java.lang.Math class?
- Why do you think the methods in the Math class are static? Could they be written as instance methods?
- In a program that uses 73.28166 in ten different places, give reasons why it makes sense to associate this value with a variable name?