Some miscellaneous concepts

Static Variables & Methods, Javadoc and Calculated Data

Produced by:

Ms. Mairéad Meagher
Dr. Siobhán Drohan
Ms Siobhan Roche

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

Department of Computing and Mathematics
http://www.setu.ie/

Topic List

1. Static Variables

2. Static Methods

3. Javadoc

4. Storing calculated data

Instance vs **Static** (Class) Variables

Instance

Multiple objects created from the same class blueprint,

- each have their own distinct copies of ***instance variables***.

Static

For variables that are common to all objects (instances)

- Use the **static** modifier.

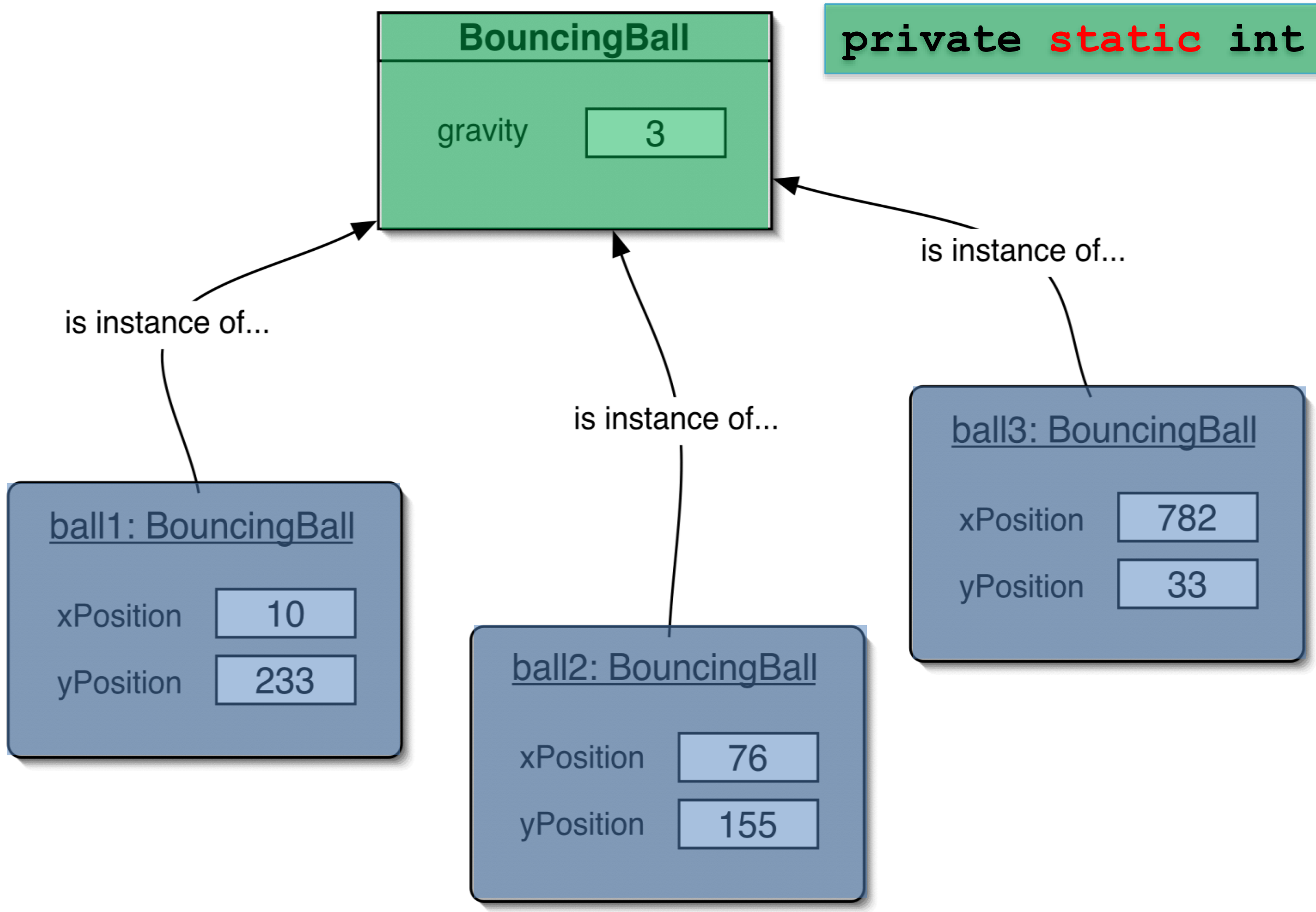
Fields that have
the static modifier in their declaration are called:

- ***static fields***
- or
  class variables.

https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

Instance vs **Static** (Class) Variables

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



CONSTANTS

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

- **`private`** : access modifier, as usual
- **`static`** : class variable
- **`final`** : constant (cannot change the value).

*Naming standard for final fields is ALL CAPITALS.

Topic List

1. Static Variables
2. Static Methods
3. Javadoc
4. Storing calculated data

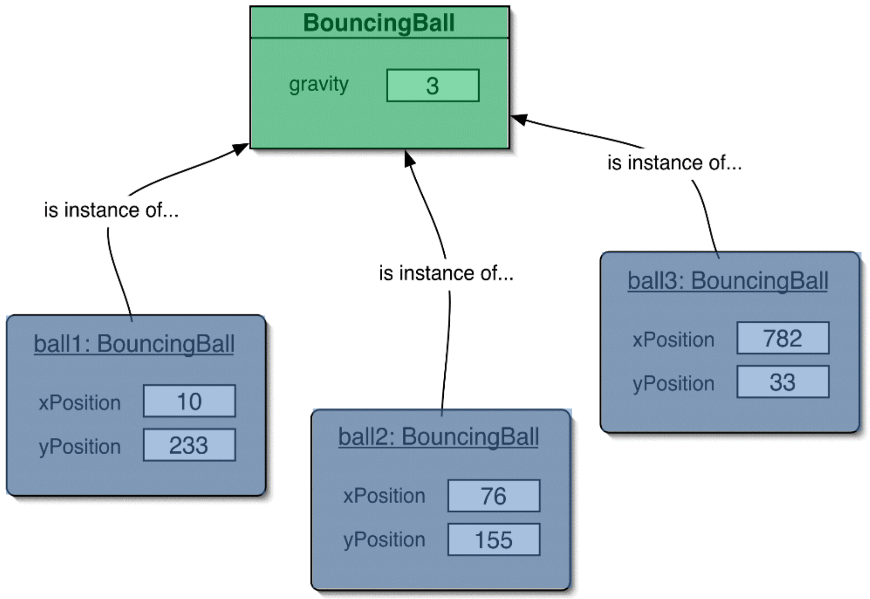
Static Methods

- Java supports **static methods**
  as well as static variables.

- Static methods
  - have the **static** modifier in their declarations
  - should be **invoked with the class name**,
    without the need for creating an instance of the class, as in:

```
ClassName.methodName(args)
```

Static Methods



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

```
public static int getGravity()
{
    return gravity;
}
```

A common use for static methods is **to access static fields.**

- E.g. we could add a **static method** to the **BouncingBall** class to access the **gravity static field**:

Topic List

1. Static Variables
2. Static Methods
3. Javadoc
4. Storing calculated data

Writing class **documentation**

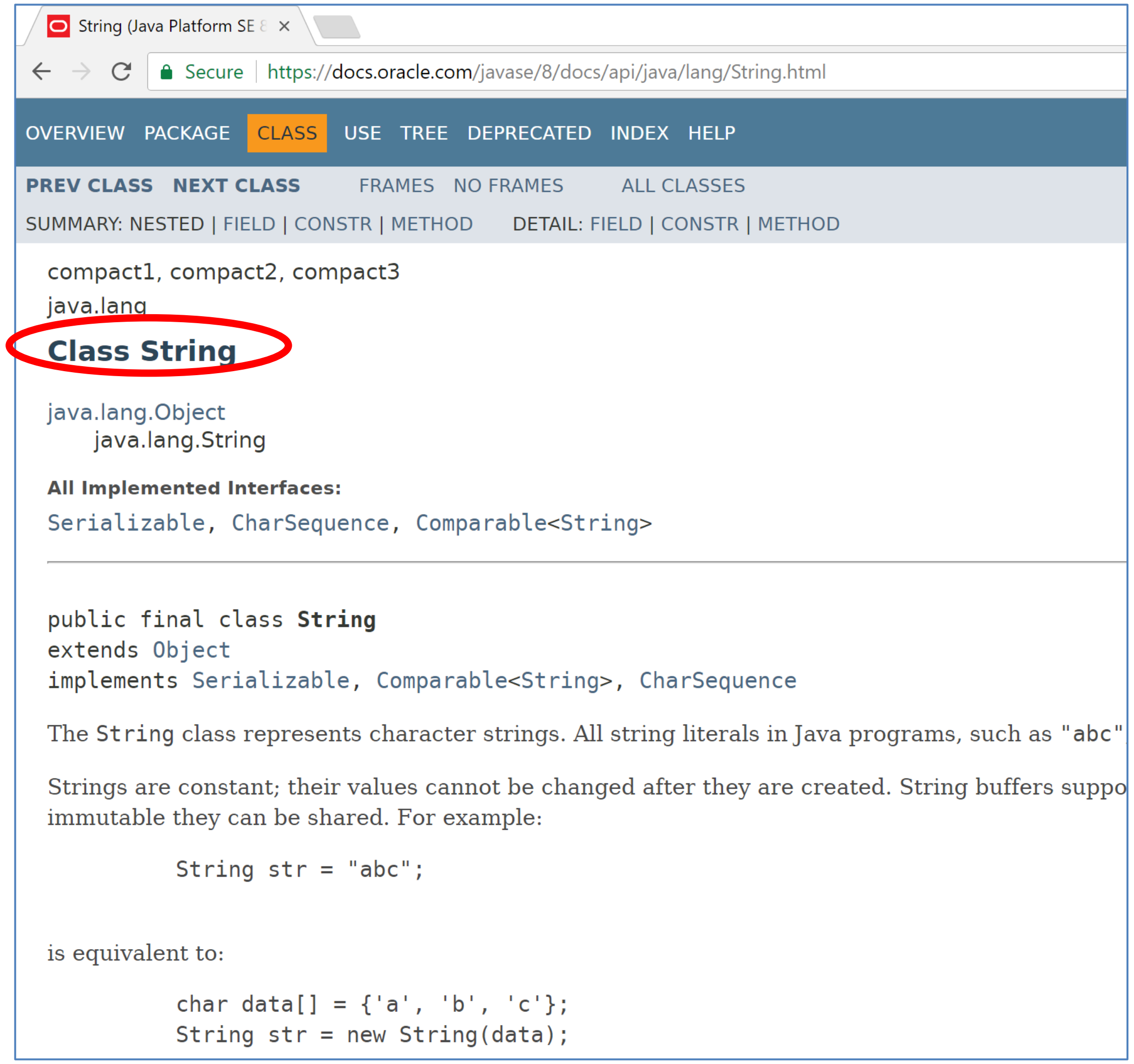
- Your own classes should be documented the same way library classes are.

- Other people should be able to use your class without reading the implementation.

- Make your class a 'library class'!

Example of Library Documentation

Java Documentation



String (Java Platform SE 8 ×

Secure | https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

`Serializable, CharSequence, Comparable<String>`

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

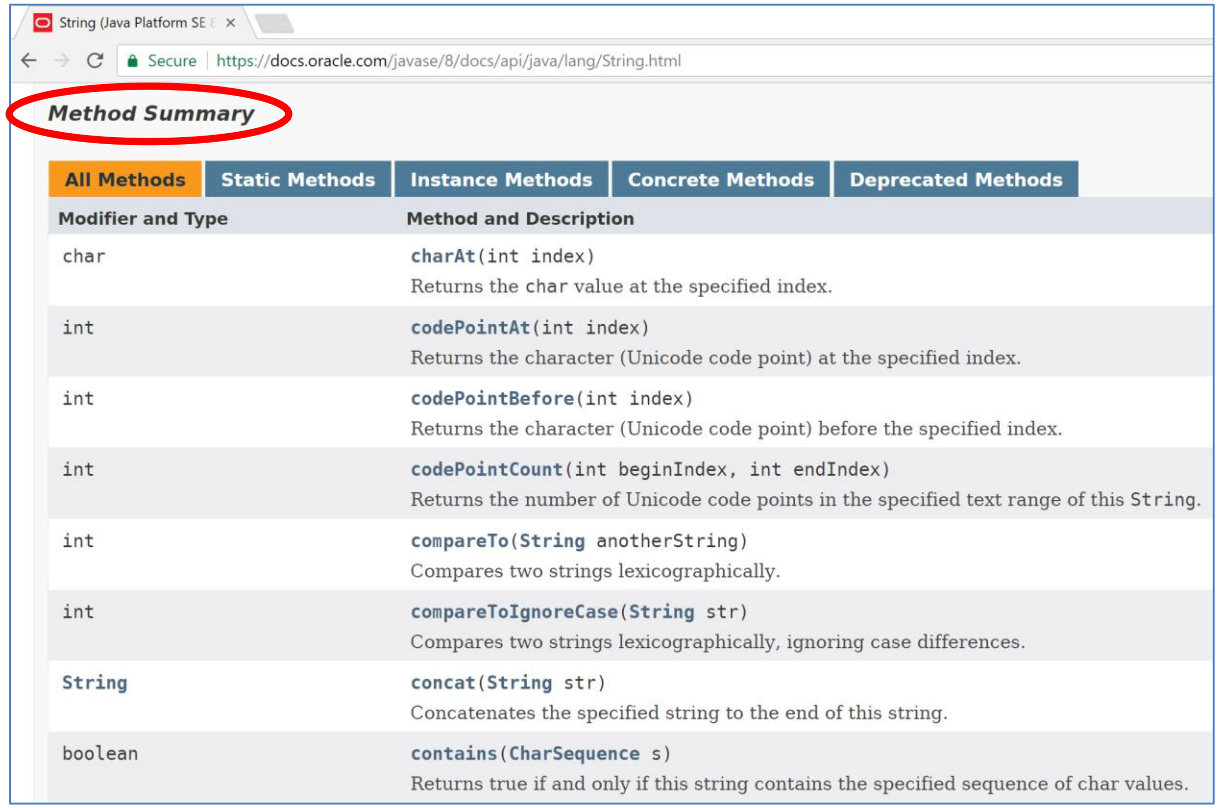
The `String` class represents character strings. All string literals in Java programs, such as `"abc"`

Strings are constant; their values cannot be changed after they are created. String buffers suppo
immutable they can be shared. For example:

```
     String str = "abc";
```

is equivalent to:

```
     char data[] = {'a', 'b', 'c'};
     String str = new String(data);
```



String (Java Platform SE 8 ×

Secure | https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method and Description
`char`	`charAt(int index)` Returns the `char` value at the specified index.
`int`	`codePointAt(int index)` Returns the character (Unicode code point) at the specified index.
`int`	`codePointBefore(int index)` Returns the character (Unicode code point) before the specified index.
`int`	`codePointCount(int beginIndex, int endIndex)` Returns the number of Unicode code points in the specified text range of this `String`.
`int`	`compareTo(String anotherString)` Compares two strings lexicographically.
`int`	`compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.
`String`	`concat(String str)` Concatenates the specified string to the end of this string.
`boolean`	`contains(CharSequence s)` Returns true if and only if this string contains the specified sequence of char values.

Elements of documentation - **class**

Documentation for a <u>class</u> should include:

- **class name**
- **comment** describing the overall purpose and characteristics of the class
- **version number**
- **author**s' names
- **constructor** documentation (for all constructors)
- **method** documentation (for all methods)

Elements of documentation - **methods**

Java™ Documentation

The documentation for each ***constructor*** *and* ***method*** *should include:*

- **method name**
- **return type**
- **return value** description
- **method purpose** and function description
- **parameter names and types**
- **parameter** description (for each parameter)

Javadoc

- **javadoc comment** - start symbol:

  /**

- Immediately before a…

  - **class declaration** is read as a *class comment*.
  - **method signature** is read as a *method comment*.

- Other special key symbols for formatting documentation include:

  @version
  @author
  @param
  @return

Javadoc

Class comment:

```
/**
 * The Responder class represents a response
 * generator object. It is used to generate an
 * automatic response.
 *
 * @author     Michael Kölling and David J. Barnes
 * @version    1.0  (30.Mar.2006)
 */
```

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)
{
   ...
}
```

Topic List

1. Static Variables
2. Static Methods
3. Javadoc
4. Storing calculated data

The danger lurking within!

Calculated data

```
public class Employee
{
     private double salary;
     private double deductions;
     private double netSalary;
     //
     //

     public void   calculateNetSalary()
     {
        netSalary = salary - deductions;
     }

     public void setSalary(double salary)
     {
        this.salary = salary;
     }
}
```

netSalary is calculated data.

!!! DATA INTEGRITY WARNING !!!

- **netSalary field** can contain **stale data.**
- Don't store **netSalary** in a field
- Calculate this when needed instead
- **calculateNetSalary()**

NB: **setSalary()**
doesn't recalculate the net salary?

Calculated data

```
public class Employee
{
     private double salary;
     private double deductions;
     //

    public double calculateNetSalary()
    {
          return (salary - deductions);
    }

     public void setSalary(double salary)
     {
          this.salary = salary;
     }
}
```

netSalary field

- is no longer declared.

calculateNetSalary()

- now returns the result of the calculation.

No calculated data is stored, so **no stale data**!

Summary

1. Static Variables
   - Class variables
   - Shared between multiple instances
   - Add final turns it into a CONSTANT

2. Static Methods
   - Used for accessing static variables

3. Javadoc
   - Modifying comments means we can run the Javadoc compiler on our code to generate the documentation similar to Java library documentation

4. Storing calculated data
   - Don’t!
   - Write a method instead to calculate at runtime
   - Avoids STALE data

Any Questions?