

# Topic 6: Library Classes

Programming Practice and Applications (4CCS1PPA)

---

Dr. Martin Chapman  
Thursday 27th October

[programming@kcl.ac.uk](mailto:programming@kcl.ac.uk)  
[martinchapman.co.uk/teaching](http://martinchapman.co.uk/teaching)

To understand how **existing classes** can be used to to introduce **additional functionality** into our programs.

- To understand the meaning of the **static** keyword.

# Preface

---

## REMEMBER: METHOD INPUT (1)

When considering method input (as opposed to just method output; what a method returns), we asked the following question:

What if I didn't just want to print `Martin` in a box to the screen, but any **number**, an arbitrary number of times?

```
System.out.println("+-----+");  
System.out.println("|    ?    |");  
System.out.println("+-----+");
```

## REMEMBER: METHOD INPUT (2)

This lead to the introduction of the `printNumber` method, with a single integer parameter.

```
public class NumberPrinter {  
  
    public void printNumber(int num) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + num + "|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

## REMEMBER: METHOD INPUT (3)

What if we wanted to go through a similar process for a name printer class (i.e. a generalised version of MartinPrinter)?

```
public class NamePrinter {  
  
    public void printName(    ?    name) {  
  
        System.out.println("+-----+");  
        System.out.println("|" + name + "|");  
        System.out.println("+-----+");  
  
    }  
  
}
```

Which datatype  
should we  
employ here?

# STRINGS (1)

It's clear that we need some form of **text** datatype.

The formal name for this datatype is a **String**.

```
public class NamePrinter {  
    public void printName(String name) {  
        System.out.println("+-----+");  
        System.out.println("|" + name + "|");  
        System.out.println("+-----+");  
    }  
}
```

## STRINGS (2)

We have seen lots of String **literals** so far, but we haven't yet seen the String **datatype**.

```
String martin = "Martin";
```

A variable of the String datatype is declared and assigned in a familiar way (type followed by name with the optional assignment of a literal), so why didn't I discuss String with the **other** primitive types?

We can answer this question by first considering a problem...



# STORING PINS (1)

In the previous topic (Exercise Topic5-2), I asked you to consider how to ensure that each account requires a different pin.

```
public class BankAccount {
```

```
    private double balance;
```

```
    private int pin;
```

```
    public BankAccount(double balance, int pin) {
```

```
        this.balance = balance;
```

```
    }
```

```
    public void withdraw(double amount, int pin) {
```

```
        if ( pin == this.pin ) {
```

```
            balance = balance - amount;
```

```
        } else {
```

```
            System.out.println("Ah ah ah, you didn't say the magic word.");
```

```
        }
```

```
    }
```

*You probably implemented something along these lines.*

RISK ASSESSMENT —

## As you read this, teen social site is leaking millions of plaintext passwords

i-Dressup operators fail to fix bug that exposes up to 5.5 million credentials.

DAN GOODIN (US) - 27/9/2016, 07:05

## 13 Million Passwords Appear To Have Leaked From This Free Web Host - UPDATED



## Online Ad Service ClixSense Hacked; 6M Plain-Text Passwords Leaked

By *Waqas* on September 16, 2016 Email @hackread **HACKING NEWS** **LEAKS**

# STORING PINS (2): SIMPLE ENCRYPTION

Rather than storing a user's pin in **plaintext** it makes sense that we should add some simple **encryption** to that pin.

```
public class BankAccount {  
    public void withdraw(double amount, int pin) {  
        if ( pin == this.pin - 1000 ) {  
            balance = balance - amount;  
        } else {  
            System.out.println("Ah ah ah, you didn't say the magic word.");  
        }  
    }  
}
```

For example we might add 1000 to it when storing it, then subtract 1000 from it when testing the pin.

# STORING PASSWORDS (1)

Now that we know about strings, we could evolve our `BankAccount` class into an `OnlineBankAccount`, which requires a user to supply a **password** for entry, rather than a pin.

The first modification to the standard `BankAccount` class should be clear:

```
public class OnlineBankAccount {  
    private String password;  
  
    public OnlineBankAccount(String password) {  
        this.password = password;  
    }  
}
```

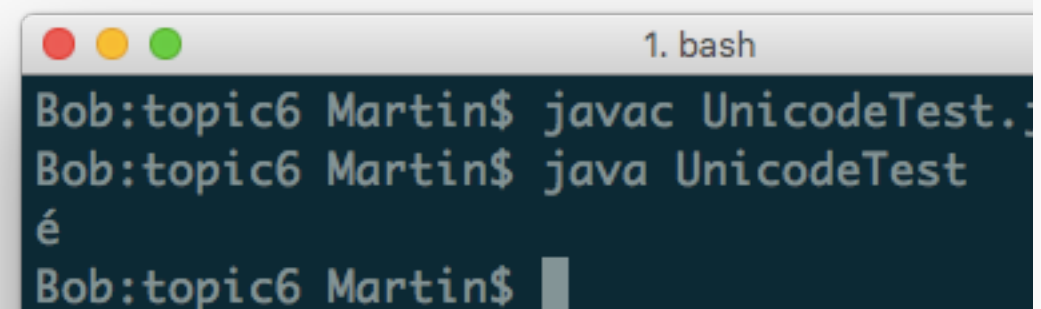
## ASIDE: WHICH CHARACTERS CAN I PLACE IN A STRING?

The use of passwords, which are often required to contain **special characters**, prompts the question: which characters can Java strings hold?

Java string support **UTF-16**, which is an encoding type that captures a **large** range of characters.

- Still, be careful when adding special characters into your program.
- If the Java compiler complains, it may be worth replacing these characters with **escaped unicode hex values**.

```
System.out.println("\u00E9");
```



```
1. bash
Bob:topic6 Martin$ javac UnicodeTest.java
Bob:topic6 Martin$ java UnicodeTest
é
Bob:topic6 Martin$
```



# STORING PASSWORDS (2): SIMPLE ENCRYPTION (1)

What about encrypting String passwords, in the same way that we did with the numeric pin?

It might be a nice idea to **reverse** the password, to avoid storing it in plaintext.

How would we go about doing this?

## STORING PASSWORDS (2): SIMPLE ENCRYPTION (2)

In (slightly more descriptive) pseudocode, we want to do something similar to the following:

*We should first capture this functionality in a method.*

```
public String encrypt(String password) {
```

*Declare an empty string*

*Starting at the end of the supplied password, go backwards through each character in that password:*

*Take the appropriate character and add it to the empty string.*

```
}
```

# STORING PASSWORDS (2): SIMPLE ENCRYPTION (3)

Declaring the empty String is straight forward.

- If we want an empty String, we must give our String an **empty literal** as, like primitive types, Strings **do not have default values**.

```
public String encrypt(String password) {
```

```
    String reversedString = "";
```

*Starting at the end of the supplied password,  
go backwards through each character in that  
password:*

*Take the appropriate character and add it  
to the empty string.*

```
}
```



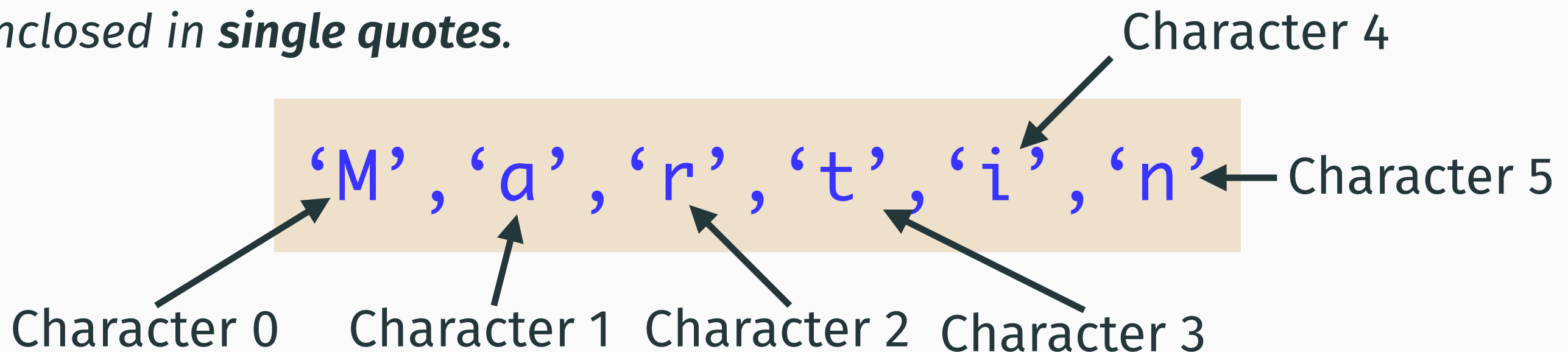
# ASIDE: STRINGS AND CHARACTERS

Strings are actually **abstractions** over **collections** of **characters**.

- This will be a familiar notion to those with a C/C++ background.

As such, we can imagine Strings in the following way:

*In Topic 2 (Topic2-2) you should have found that character literals are enclosed in **single quotes**.*



## ASIDE: INDEX VS. LENGTH

Because of this breakdown, we can imagine each character in a string as having an **index**.

- This means that there is a distinct difference between the **length** of a string and the **last index**.

‘M’, ‘a’, ‘r’, ‘t’, ‘i’, ‘n’ ← Character 5

Length = 6.

We can leverage this idea in order to approach our current problem.

## STORING PASSWORDS (2): SIMPLE ENCRYPTION (4)

We define a loop with the first index value equal to the length of the string minus 1 (remember index vs. length) and then go **backwards** through that loop.

```
public String encrypt(String password) {  
    String reversedString = "";  
    for ( int i = String length - 1; i >= 0; i-- ) {  
        Take the appropriate character and add it  
        to the empty string.  
    }  
}
```

*How do we do these two things?*

## STORING PASSWORDS (2): SIMPLE ENCRYPTION (5)

We use the value of the index within the loop to perform a **repeated operation**: reading an individual character from the password string and adding it to our string variable.

```
public String encrypt(String password) {  
    String reversedString = "";  
    for ( int i = password.length() - 1; i >= 0; i-- ) {  
        reversedString += password.charAt(i);  
    }  
    return reversedString;  
}
```

*Gives us the length of the string.*

*Gives us the character at a given index in the string.*

# STORING PASSWORDS (2): SIMPLE ENCRYPTION (5)

`password.length() - 1`

`"Martin"`

`i`

`charAt(i)`

`reversedString`

5

`'n'`

`"n"`

4

`'i'`

`"ni"`

3

`'t'`

`"nit"`

2

`'r'`

`"nitr"`

1

`'a'`

`"nitra"`

0

`'M'`

`"nitraM"`

## ASIDE: BACK TO CONCATENATION

Remember we've seen two uses of the plus symbol.

```
System.out.println("|" + num + "|");
```

```
balance = balance + amount;
```

The former we referred to as **concatenation**.

We can now formalise the rule for when concatenation occurs: if **either** of the operands to an addition are strings, the operation will be treated as a concatenation.

```
reversedString += password.charAt(i);
```

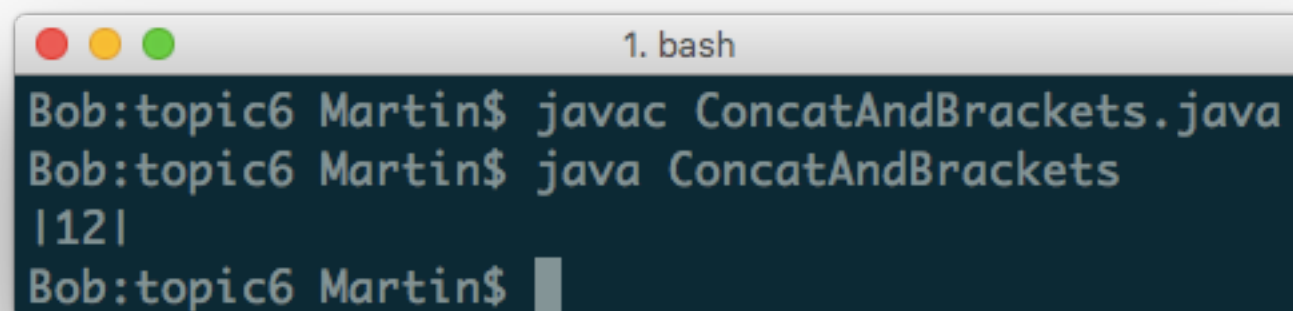
```
reversedString = "ni" + "t";
```

*Here we see concatenation again.*

## ASIDE: BRACKETS AND CONCATENATION

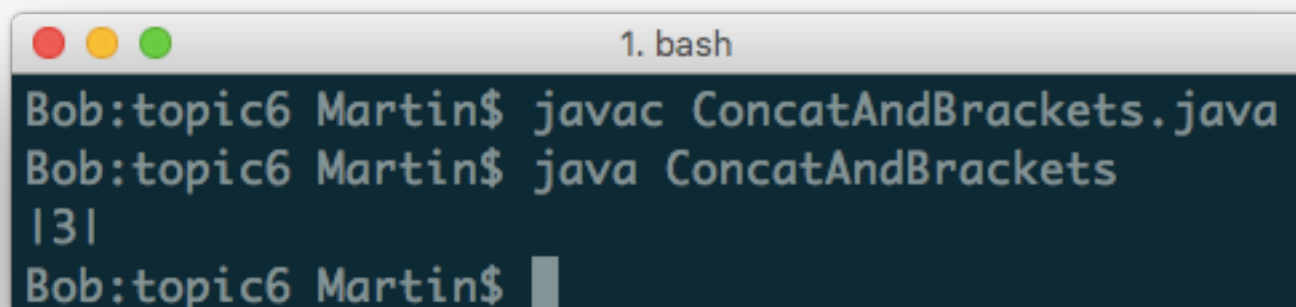
If we want to overwrite this default behaviour, in order to, say, print the result of some arithmetic, we can do so using **brackets**.

```
System.out.println("|" + 1 + 2 + "|");
```



```
1. bash
Bob:topic6 Martin$ javac ConcatAndBrackets.java
Bob:topic6 Martin$ java ConcatAndBrackets
|12|
Bob:topic6 Martin$
```

```
System.out.println("|" + (1 + 2) + "|");
```



```
1. bash
Bob:topic6 Martin$ javac ConcatAndBrackets.java
Bob:topic6 Martin$ java ConcatAndBrackets
|3|
Bob:topic6 Martin$
```

# STRING: PRIMITIVE TYPE OR CLASS?

```
password.length()
```

```
password.charAt(i);
```

These looks like method calls, which very much suggests that this string variable is an **object**.

But why, then, do we define a new string like a **primitive type**...

```
String martin = "Martin";
```

...instead of like a class type?

```
String martin = new String("Martin");
```

*(Although this works too; we'll see this shortly.)*

**Is String a primitive type or a class?**



# STRINGS (3)

I didn't discuss strings along with the other primitive types because the answer to this question is complex.

String variables can be declared and assigned in the same way as primitive type variables, but, behind the scenes, the String type is actually a **class** (**open question: where does this class come from?**), and String variables thus contain copies of this class.

- This allows us to declare variables of a commonly used datatype, and construct literals of this type, **effectively**, without having to use the new command.
  - We can use the String type in **typical** places (because we've learnt that class types can replace primitive types).
- **More importantly, when we don't explicitly request a copy of the String class, we allow the JVM to decide whether to give us a new copy or not.**

# COPYING STRINGS (1)

```
public void deposit(double amount) {  
    balance = balance + amount;  
    System.out.println("Your current balance is: " + balance);  
}
```

```
public void withdraw(double amount) {  
    balance = balance - amount;  
    System.out.println("Your current balance is: " + balance);  
}
```

It's often the case that we use the same strings throughout a class.

# COPYING STRINGS (2)

```
public class Driver {  
    public static void main(String[] args) {  
  
        Pair pairA = new Pair();  
        Pair pairB = new Pair();  
  
        pairA.setPair(1, 2);  
        pairB.setPair(3, 4);  
  
        System.out.println(pairA.getValueA());  
        System.out.println(pairB.getValueA());  
  
    }  
}
```

```
public class Pair {  
  
    private int valueA;  
    private int valueB;  
  
    public void setPair(int pass  
  
        valueA = passedValueA;  
        valueB = passedValueB;  
  
}
```

```
public class Pair {  
  
    private int valueA;  
    private int valueB;  
  
    public void setPair(int pass  
  
        valueA = passedValueA;  
        valueB = passedValueB;  
  
}
```

We are familiar with the convention of storing **different copies** of a class as objects.

# COPYING STRINGS (3)

```
public static void main(String[] args) {  
    String a ← new String("A");  
    String b ← new String("A");  
    String c ← new String("A");  
    String d ← new String("A");  
}
```

'A' ← Field in String class

String class copy

'A'

'A'

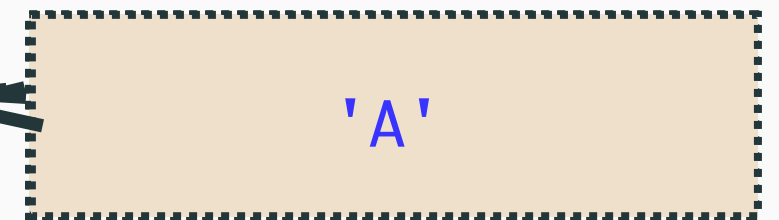
'A'

If it were also the **convention** to declare strings in this way, any duplicate Strings would be **separate copies**.

- This is, naturally, inefficient.

# COPYING STRINGS (4)

```
public static void main(String[] args) {  
    String a = "A";  
    String b = "A";  
    String c = "A";  
    String d = "A";  
}
```



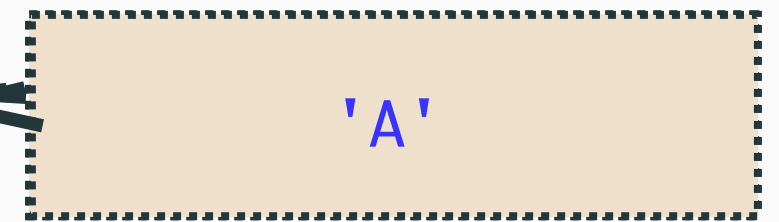
*We can view the use of a literal here as an **abstraction** over the new command.*

So, instead, the convention is that we don't **explicitly** use the new command, but instead allow the JVM to provide us with new copies of the String class at its discretion, depending on the string **literal used**.

- The JVM can thus choose to optimise by **giving us one copy**, which is then **shared amongst all the variables**, when **all of the strings requested are identical**.

# COPYING STRINGS (5)

```
public static void main(String[] args) {  
    String a = "A";  
    String b = "A";  
    String c = "A";  
    String d = "A";  
}
```



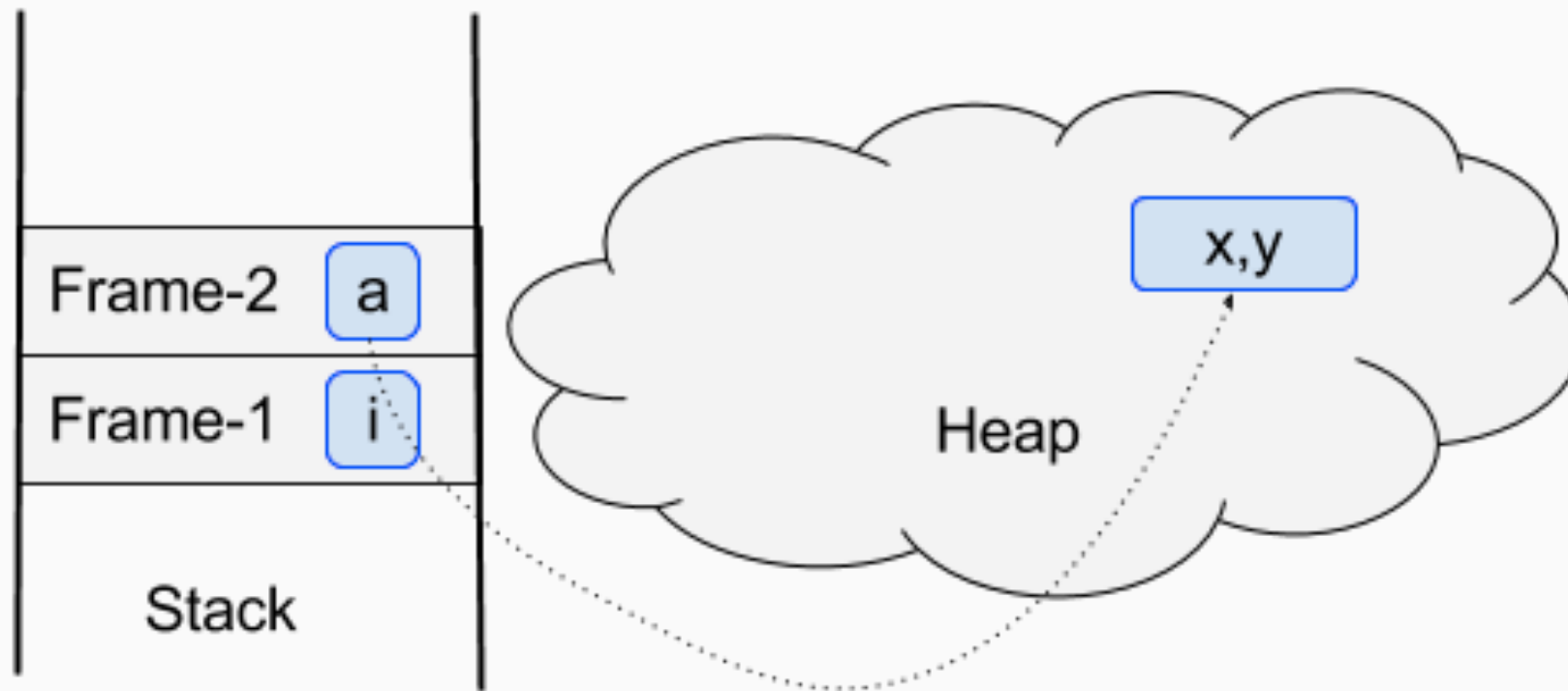
In order to avoid global changes to the shared copy, strings are **immutable** (opposite of a mutable class, where the state (fields) can be changed by mutator methods).

**Open question** (for next semester): Can we share copies of our own classes?



# REMEMBER: OBJECTS IN MEMORY

We will use the definition of an object quite loosely in the first semester of PPA, in order to **simplify** things, and help your initial understanding.



*We'll look at this in more detail next semester.*

In reality, unlike primitive values, objects are stored in another area of the JVM's memory known as the **heap**. The variable containing the copy of the class is actually a **memory reference** from a variable on the stack to an object on the heap.

## ASIDE: INTERACTING WITH A CLASS COPY WITHOUT A VARIABLE

There are several other phenomena that show that we're simplifying things at this stage.

```
copyOfMartinPrinter.printMartin();
```

```
password.length()
```

```
new MartinPrinter().printMartin();
```

```
"mypassword".length();
```

For example, we can interact with copies of a class without placing that copy into a variable.





# BACK TO COPYING STRINGS: ASKING ABOUT COPIES

```
public static void main(String[] args) {
```

```
    String a = "A";
```

```
    String b = "A";
```

```
    String c = "A";
```

```
    String d = "A";
```

```
    System.out.println(a == b);
```

```
}
```

*We ask the JVM, `are you sharing the same string copy between a and b'?*

*Are a and b **equivalent in memory**?*

*This answers one of our open questions about the **flexibility** of relational operators.*

In order to confirm that the JVM is sharing copies between our variables, we can use the familiar **comparison** operator.

- This will print **true**.

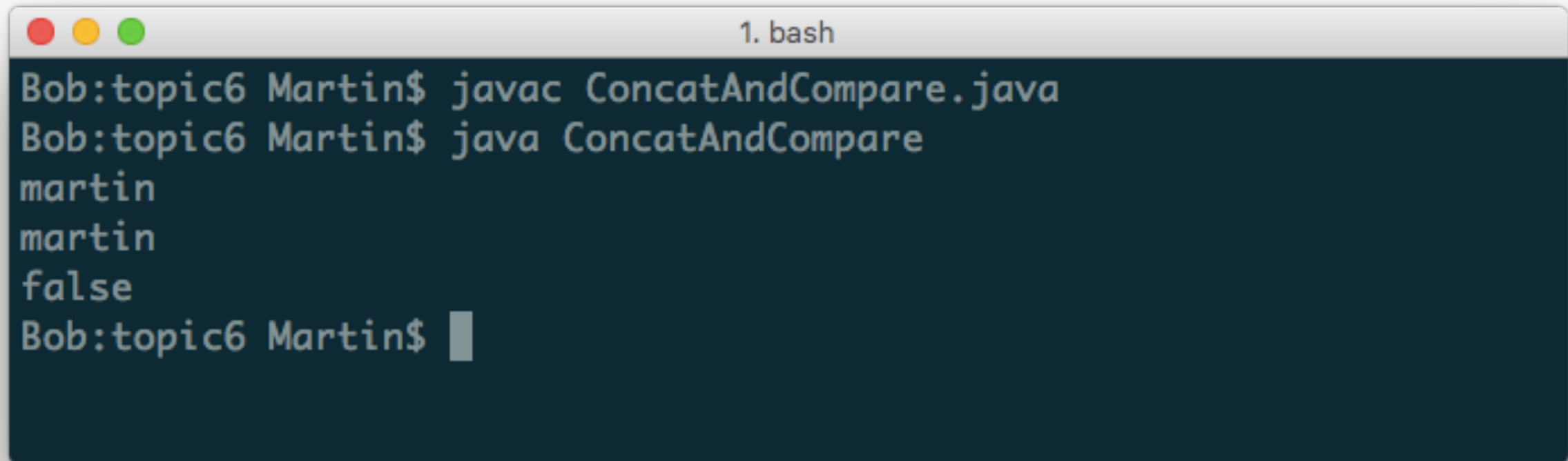
# COMPARING STRINGS (1)

There are lots of **benefits** to allowing the JVM control over the management of string copies.

However there are also drawbacks. **What will the following print?**

```
public static void main(String[] args) {  
  
    String martin = "martin";  
    String mart = "mart";  
    String in = "in";  
  
    System.out.println(mart + in);  
    System.out.println(martin);  
    System.out.println(martin == mart + in);  
  
}
```

## COMPARING STRINGS (2)

A terminal window titled "1. bash" with three colored window control buttons (red, yellow, green) in the top-left corner. The terminal shows the following text:

```
Bob:topic6 Martin$ javac ConcatAndCompare.java
Bob:topic6 Martin$ java ConcatAndCompare
martin
martin
false
Bob:topic6 Martin$
```

This occurs because the JVM stores the strings `"mart"` and `"in"` **separately even** when they are concatenated, and are thus effectively the same string.

So, although the equality operator might **look** like a valid way to compare strings, it is not.

Really we want to **examine** the **content** of the strings when checking their equality rather than simply asking about the **organisation of memory**.

So, the past few slides have been a roundabout way of saying **when you compare strings, you are advised to do so in the following way:**

## COMPARING STRINGS (4)

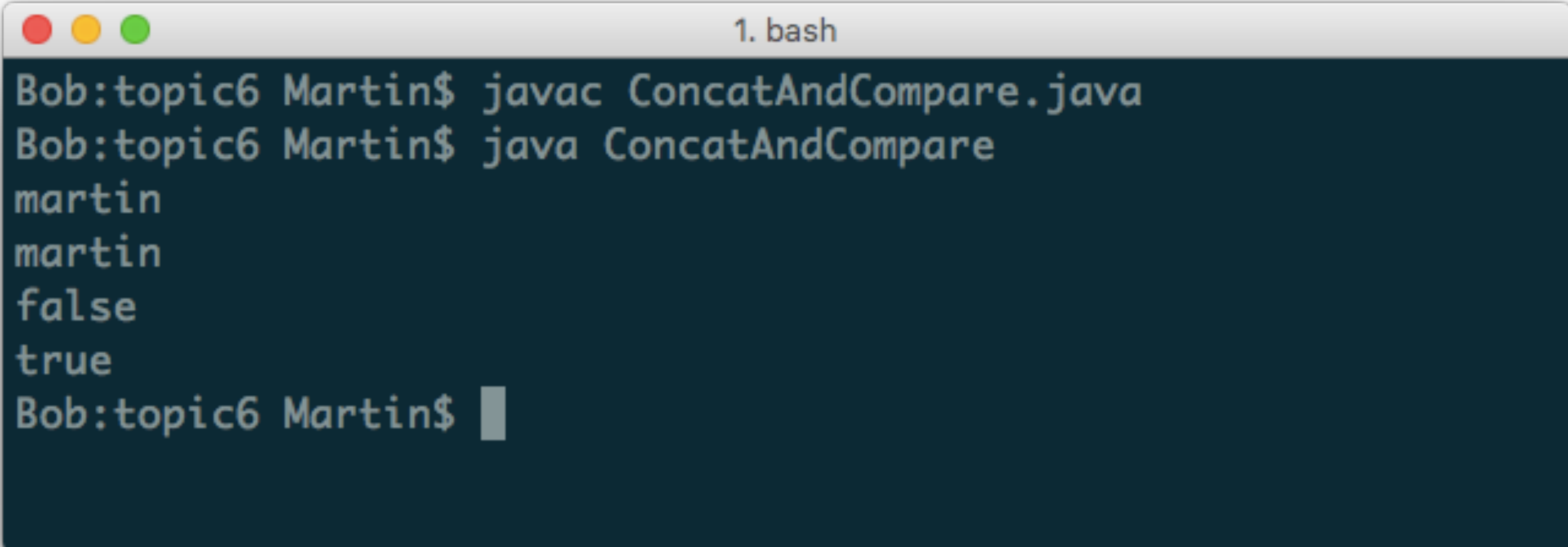
We use the **.equals** method in the String class\* to compare strings.

*\*it actually exists elsewhere, more to come.*

```
public static void main(String[] args) {  
  
    String martin = "martin";  
    String mart = "mart";  
    String in = "in";  
  
    System.out.println(mart + in);  
    System.out.println(martin);  
    System.out.println(martin == mart + in);  
    System.out.println(martin.equals(mart + in));  
  
}
```



# COMPARING STRINGS (5)



```
1. bash
Bob:topic6 Martin$ javac ConcatAndCompare.java
Bob:topic6 Martin$ java ConcatAndCompare
martin
martin
false
true
Bob:topic6 Martin$
```

# BACK TO OUR ONLINE BANK ACCOUNT

Given this notion, we can now return to our our online bank account, and implement the correct syntax for checking that a password is correct:

```
public void withdraw(double amount, String password) {  
    if ( encrypt(password).equals(this.password) ) {  
        balance = balance - amount;  
    } else {  
        System.out.println("Ah ah ah, you didn't say the magic word.");  
    }  
}
```

Let's solve the following problem **using the methods in the String class** (for practice):

- Given a string, I want to determine if all the characters in that string are unique.



# LECTURE EXERCISE: STRING MANIPULATION (2)

One potential solution:

```
public boolean isUnique(String word) {  
    for ( int i = 0; i < word.length(); i++ ) {  
        if ( !(i == word.lastIndexOf(word.charAt(i))) ) return false;  
    }  
    return true;  
}
```

*Note the brackets here so that it's clear what the unary negation refers to.*

In the laboratories, consider the following problems involving string manipulation (**you may need to finish this topic before attempting some of these**):

- Determining whether a given string is a **palindrome** (the same word when reversed e.g. racecar).
- Determine whether a given **sentence** is a palindrome (e.g. Euston saw I was not Sue).

# TAKING A STEP BACK...

```
password.length()
```

```
password.charAt(i);
```

How did I know that this was the way in which to check for a string's length, and to extract a character from a string, respectively?

- How did I know these methods **existed**?

We also asked where does this String class come from?

The answer to the latter is the **Java library**, while the answer to the former is **because the Java library is documented**.

# Library Classes

---

# OUR CLASSES VS. LIBRARY CLASSES

So far, we've looked at how capturing our code in classes allows them to become **reusable**.

Unsurprisingly, we aren't the first people to recognise this idea.

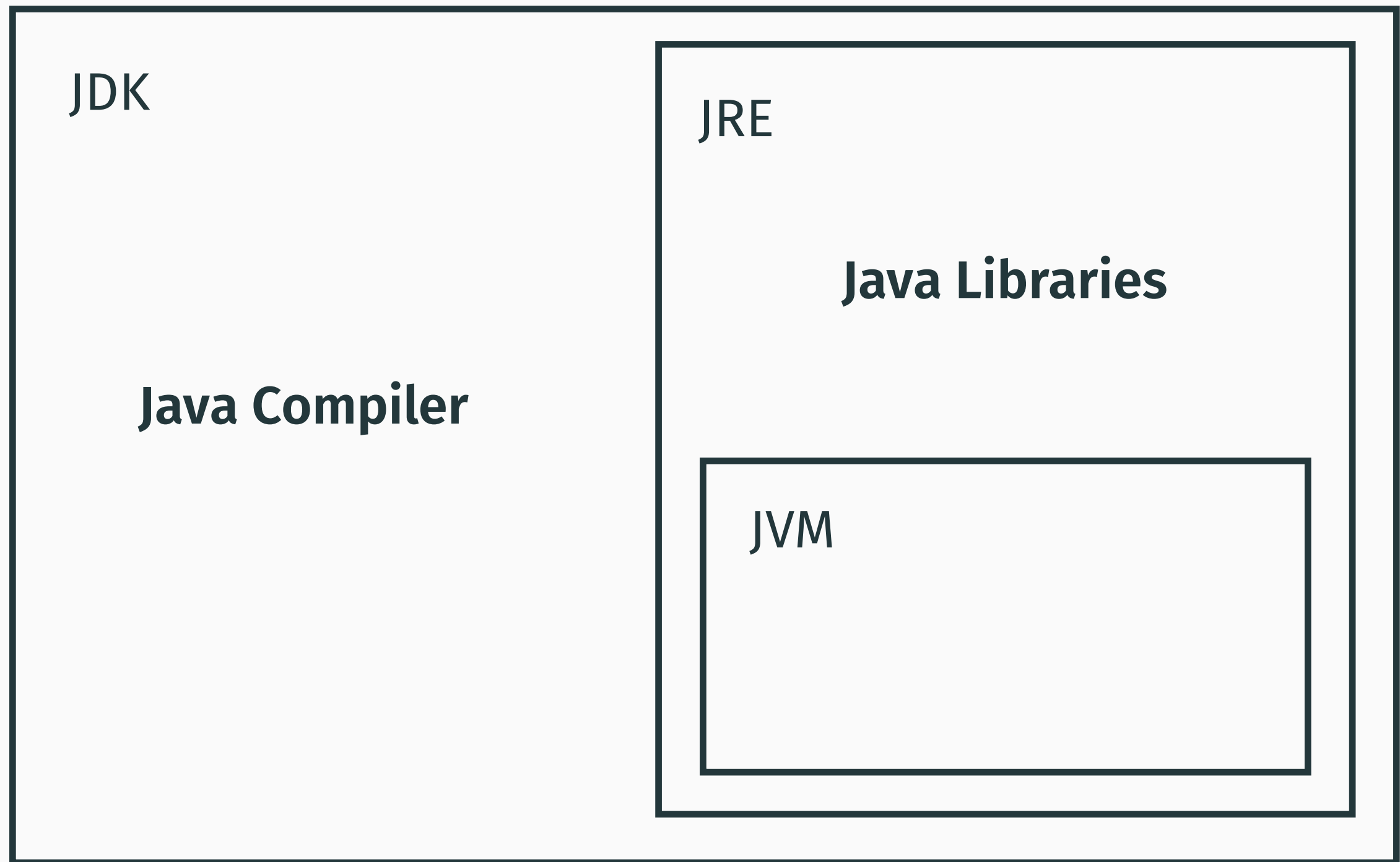
In fact, Java comes with lots of classes that **already capture existing piece of functionality**.

- **These classes are written for us by other people** (typically those who design the Java language).
- This is part of why it makes sense to teach you about **classes first**.

We call these **library classes**.

The **String** class is an example of a library class.

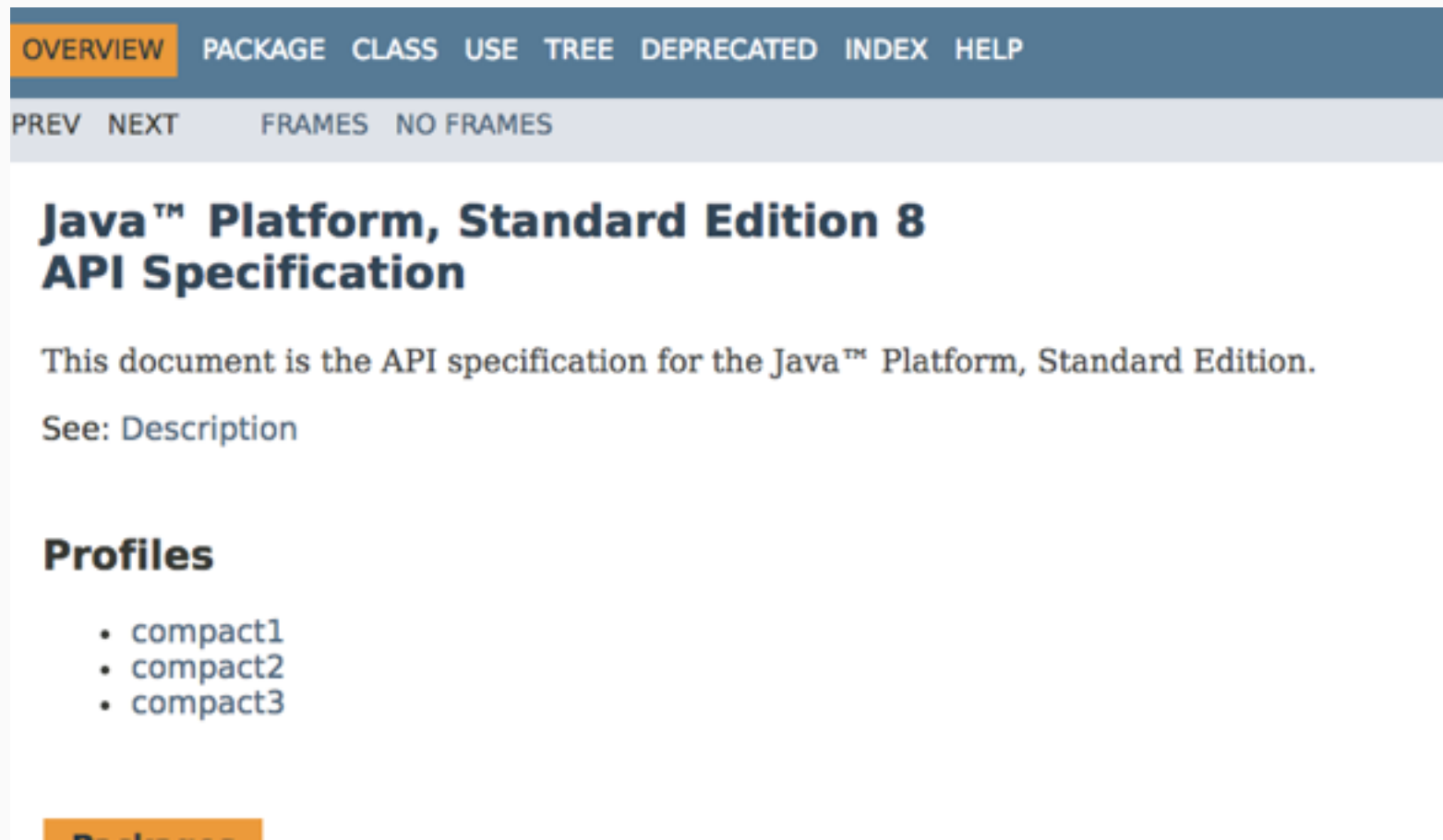
## ASIDE: JDK VS. JVM VS. JDE



Library classes exist as part of the software we use to run our programs.

To help us understand the functionality offered by library classes, Java provides us with **documentation** for each of these classes.

- This is often known as the **Java API**, but in reality the term `API' has a more comprehensive meaning.



The screenshot shows the top of the Java Platform, Standard Edition 8 API Specification website. It features a navigation bar with links: OVERVIEW (highlighted), PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. Below this is a secondary bar with links: PREV, NEXT, FRAMES, and NO FRAMES. The main heading is "Java™ Platform, Standard Edition 8 API Specification". Below the heading is a paragraph: "This document is the API specification for the Java™ Platform, Standard Edition." followed by a link "See: Description". Under the heading "Profiles", there is a bulleted list: compact1, compact2, and compact3.

*Googling `Java', followed by the name of a class typically yields the documentation for that class as the top result.*

It's because of this documentation that we are aware of the **capabilities** (i.e. the methods available) of the String class.

## Class String

### charAt

```
public char charAt(int index)
```

Returns the char value at the specified index. An index ranges from 0 to length() - 1, the next at index 1, and so on, as for array indexing.

If the char value specified by the index is a surrogate, the surrogate value is returned.

#### Specified by:

charAt in interface CharSequence

#### Parameters:

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





# REMEMBER: CONTROL 1: DOCUMENTING YOUR CODE (1)

We use documentation to help us understand our **own** code, but remember it's also a useful tool in helping **other** people to understand our code.

```
/**
 * Prints the supplied number surrounded by a box.
 */
public void printNumber(int num) {

    System.out.println("+-----+");
    System.out.println("|" + num + "|");
    System.out.println("+-----+");

}
```

Accessible code documentation is something that's **indirectly** supported by object-orientation (more later).

This doesn't stop people using your code **incorrectly**, but reduces the risk of it happening.

# BACK TO CONTROL (1)

```
/**
Bob:topic3 Martin$ javadoc NumberPrinter.java
Loading source file NumberPrinter.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_60
Building tree for all the packages and classes...
Generating ./topic3/NumberPrinter.html...
NumberPrinter.java:8: warning: no @param for num
    public void printNumber(int num) {
                          ^
Generating ./topic3/package-frame.html...
Generating ./topic3/package-summary.html...
Generating ./topic3/package-tree.html...
Generating ./constant-values.html...
Building index for all the packages and classes...
Generating ./overview-tree.html...
Generating ./index-all.html...
Generating ./deprecated-list.html...
Building index for all classes...
Generating ./allclasses-frame.html...
Generating ./allclasses-noframe.html...
Generating ./index.html...
Generating ./help-doc.html...
1 warning
Bob:topic3 Martin$
```

ended by a box.

");

















Object-orientation supports accessible documentation by encouraging us to split our code into methods, which we then comment, and these comments then feature in **generated documentation**.



# BACK TO CONTROL (2)

## Method Summary

All Methods	Static Methods	Instance Methods	Concrete Methods
Modifier and Type		Method and Description	
char		<b>charAt</b> (int index) Returns the char value at the specified index.	

Today		
	<a href="#">allclasses-frame.html</a>	
	<a href="#">allclasses-noframe.html</a>	
	<a href="#">constant-values.html</a>	
	<a href="#">deprecated-list.html</a>	
	<a href="#">help-doc.html</a>	
	<a href="#">index-all.html</a>	
	<a href="#">index.html</a>	
	<a href="#">overview-tree.html</a>	

## Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type		Method and Description
void		<b>printNumber</b> (int num) Prints the supplied number surrounded by a box.
Methods inherited from class java.lang.Object		
clone equals finalize getClass hashCode notify notifyAll toString		

We can make our own small (offline) addition to the Java documentation.

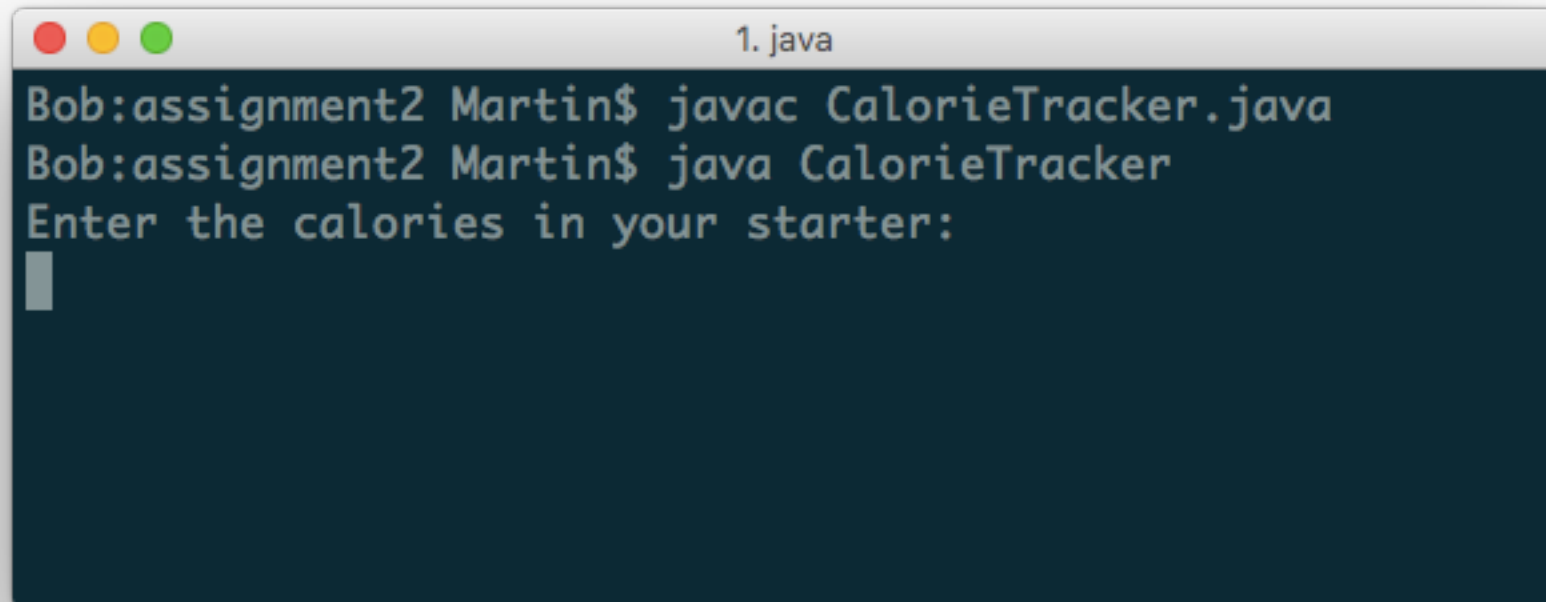


# User Input

---

# USER INPUT (1)

In order to make our programs more **interactive**, it's often important to ask a **user** for input.

A terminal window titled "1. java" with a dark background. It shows the following text: "Bob:assignment2 Martin\$ javac CalorieTracker.java", "Bob:assignment2 Martin\$ java CalorieTracker", and "Enter the calories in your starter:". A cursor is visible on the line following the prompt.

```
Bob:assignment2 Martin$ javac CalorieTracker.java
Bob:assignment2 Martin$ java CalorieTracker
Enter the calories in your starter:
█
```

*We might consider a more user friendly version of Assignment 2, which allows for calories to be input while the program is running.*

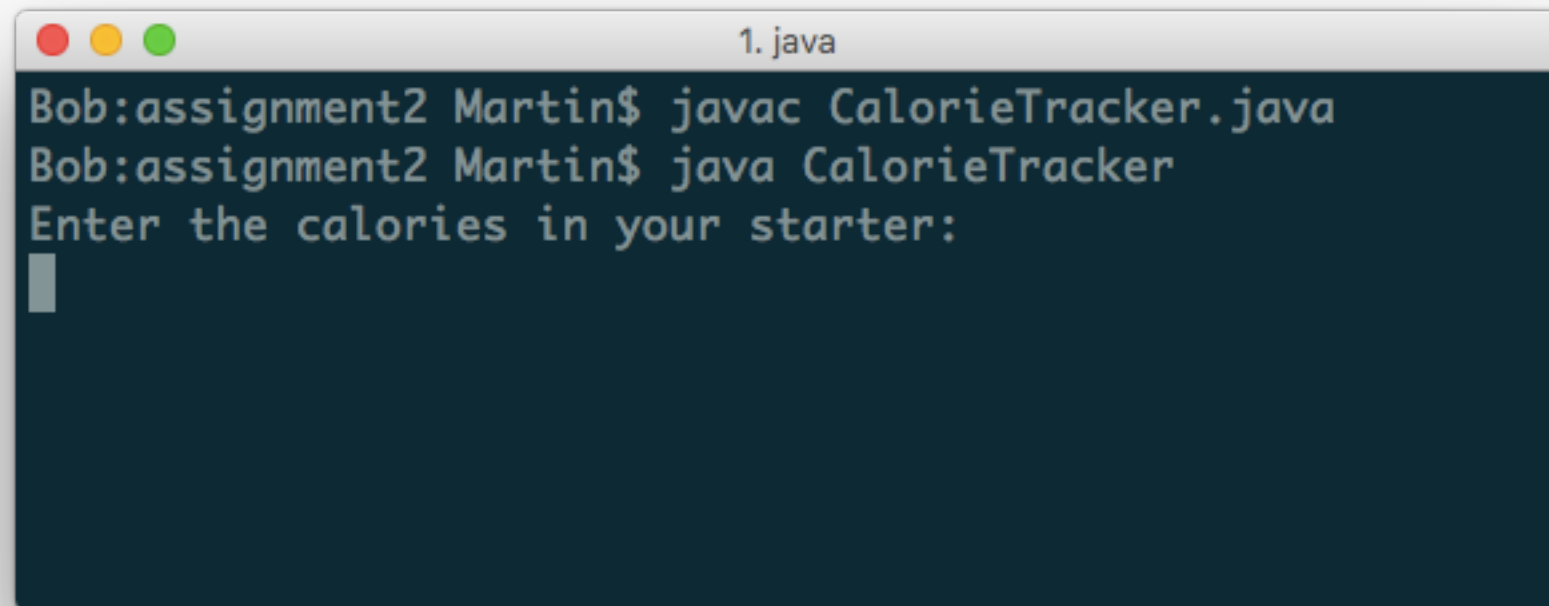
These are typically non-technical users who interact with our program **while it is running**.

We have previously considered **technical users** (e.g. other developers) who are likely to interact with our program (i.e. our classes) **while they are developing**.

```
starter.setCalories(140);
```

## USER INPUT (2)

In Java, we typically take user input **from the command line**.

A terminal window titled "1. java" with a dark blue background and light gray text. It shows the following commands and output:

```
Bob:assignment2 Martin$ javac CalorieTracker.java
Bob:assignment2 Martin$ java CalorieTracker
Enter the calories in your starter:
█
```

When user input is taken from the command line (while the program is running), the program **pauses** and waits for the user to enter information into the terminal.

This information can then be used in the program.

# USING A LIBRARY CLASS IN OUR PROGRAM

To create this behaviour ourselves would obviously be quite **difficult**, and as such it makes sense that we should look for some existing code (a library class), to help us out.

There is a distinct process to go through when we want to use a library class in our program:

- **Discover** which library class is appropriate for our needs.
- **Add** that library class to our program.
- Use the documentation to determine **how to use** that class.

Determining which library classes will fulfil our needs (in this case user input), can be done in a number of ways.

- Typical language **resources** (web searches, books, lecture notes, etc.).
- **Existing documentation.**
  - Links will often exist in the pages of the Java documentation, for example, to other classes.
- Experience from **other languages.**



For user input, the class we need is called **Scanner**.

## **Class Scanner**

```
java.lang.Object  
    java.util.Scanner
```

### **All Implemented Interfaces:**

```
Closeable, AutoCloseable, Iterator<String>
```

---

```
public final class Scanner  
extends Object  
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The tokens are then converted into values of different types using the various next methods.

For example, this code allows a user to read a number from `System.in`:

# USING A LIBRARY CLASS IN OUR PROGRAM: ADDING A CLASS

Classes are useful to us because they offer **public methods** that contain some functionality.

Remember our current rule is that any public methods within a class within our **own** program can be accessed by any **other** class within our program.

- This is true given the current way in which we are organising our classes, but will change when we add **additional** organisation next semester.

What about public methods in classes that are **not** in our program, such as Scanner?



# IMPORTING CLASSES (1)

If we want to access the methods of, or indeed make objects of, a class that existing within an **external library** (or another package; more next term), we have to **import** that class into our program.

This allows us to **reference** that class **as if it were one of our own**, and thus make objects of it, and call public methods.

- But we can't edit it **directly**.
- **We** are now the **users** of classes written by other people.

In the case of Scanner this necessitates the following import statement (at the **top** of a class):

```
import java.util.Scanner;
```

# IMPORTING CLASSES (2): DOT STRUCTURE

```
import java.util.Scanner;
```

## Class Scanner

java.lang.Object  
java.util.Scanner

**All Implemented Interfaces:**

What this **dot structure** means should become clear next semester, but for now it suffices to say that this information can be gleaned from the documentation itself.

# TRACKING A USER'S CALORIES (1)

```
import java.util.Scanner;  
  
public class CalorieTracker {  
    public static void main(String[] args) {
```

*If we want to facilitate user input in our calorie tracker program, we need to **import** the Scanner class.*

## ASIDE: WHY DIDN'T I NEED TO IMPORT STRING?

Earlier I said that String was a library class (an idea supported by the presence of documentation), but I've been using strings freely without importing them into my class.

```
import java.lang.String;
```

### Class String

```
java.lang.Object  
java.lang.String
```

I **could** import String if I wanted to, but we need strings so frequently that Java imports the String class **for us**.

- We've already seen Java doing things for us such as adding implicit constructors.

# USING A LIBRARY CLASS IN OUR PROGRAM: USE

Now that we have an external class imported, using that class typically requires us to answer two questions. We usually answer these questions using the **documentation**:

- How are objects of this external class **constructed**?
- Which **methods** are available to us once we have obtained an object of this class?
  - Which **parameters** do they require?
  - Which **values** do they return?

# CONSTRUCTING A SCANNER (1)

From the Scanner documentation, it is clear that there are multiple ways in which to construct objects of this class:

## Constructor Summary

### Constructors

#### Constructor and Description

**Scanner**(File source)

Constructs a new Scanner that produces values scanned from the specified file.

**Scanner**(File source, String charsetName)

Constructs a new Scanner that produces values scanned from the specified file.

**Scanner**(InputStream source)

Constructs a new Scanner that produces values scanned from the specified input stream.

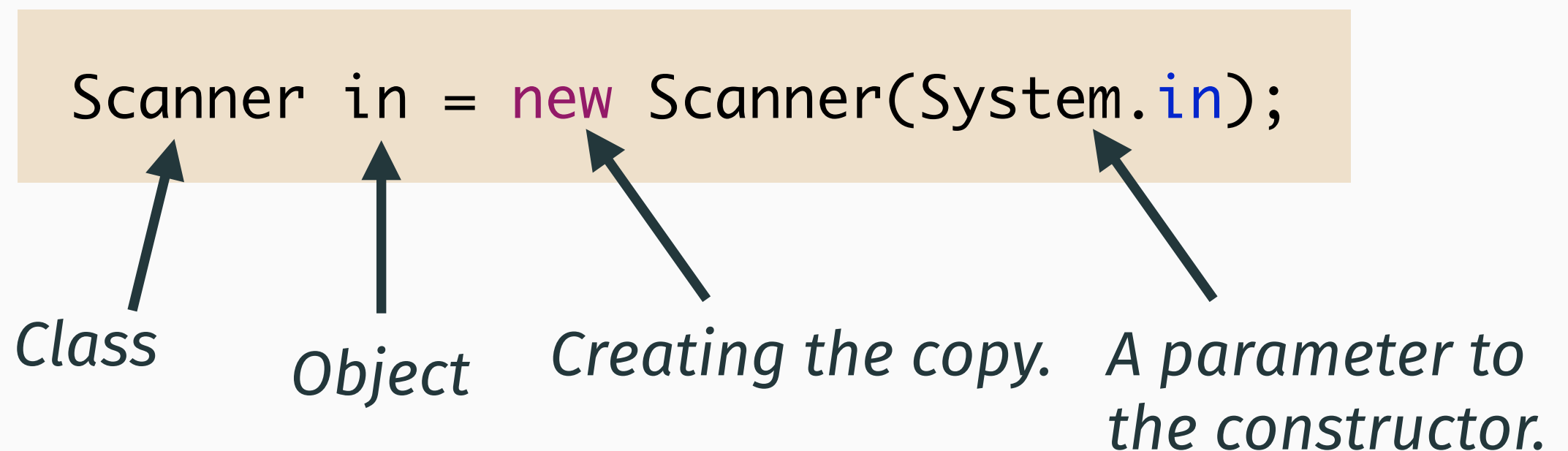
Scanner is a **versatile** class that can be used for many different purposes (some of which we'll see), but for now we'll use it solely for reading a **stream of input** from the command line.





## CONSTRUCTING A SCANNER (2)

For now, I am going to give you the code to construct a Scanner object with an stream of input as **boilerplate**, and then come back later to describe in more detail where this information comes from:



*Remember our understanding of these terms will evolve next semester.*

## TRACKING A USER'S CALORIES (2)

```
import java.util.Scanner;

public class CalorieTracker {

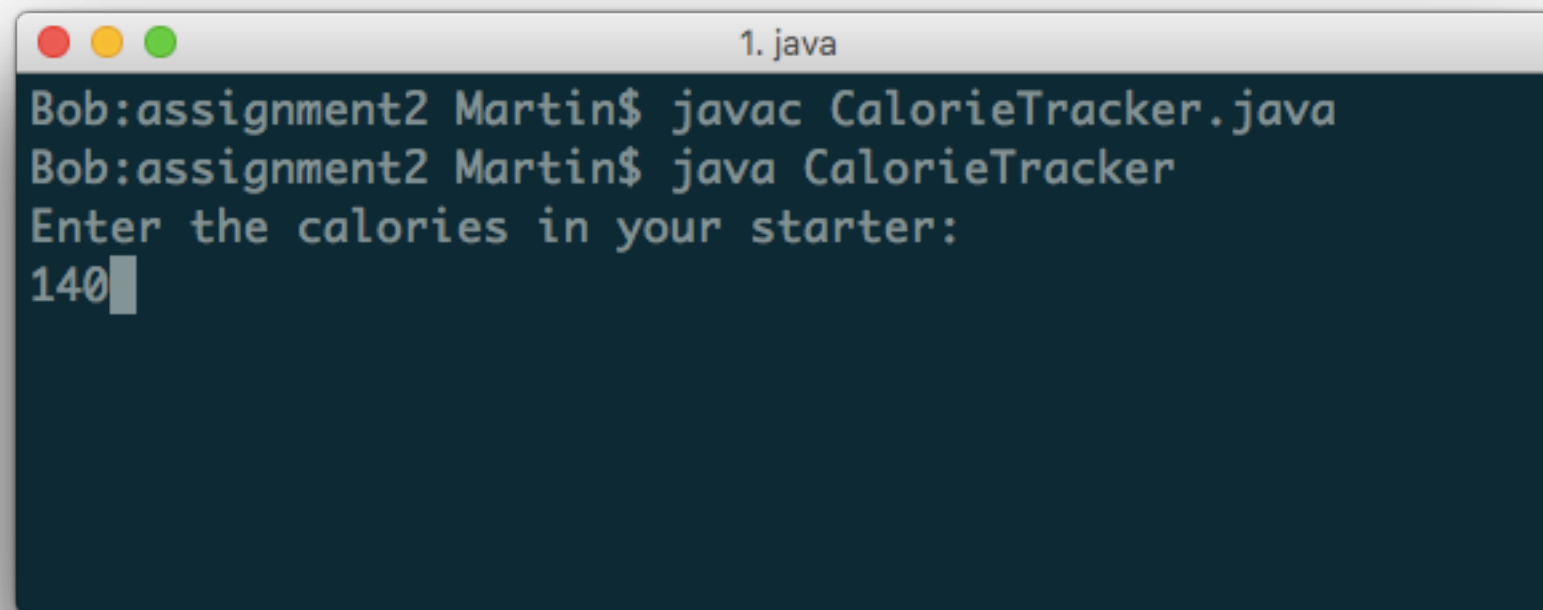
    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
```

*If we want to facilitate user input in our calorie tracker program, we have to import the Scanner class **and then** make an object of it (as we would with one of our own classes).*

# USING A SCANNER (1)

Remember we're aiming to open our CalorieTracker up to this style of user input:

A terminal window titled "1. java" with a dark blue background and white text. It shows the compilation and execution of a Java program. The prompt is "Bob:assignment2 Martin\$". The first command is "javac CalorieTracker.java". The second command is "java CalorieTracker". The program then prompts the user with "Enter the calories in your starter:". The user has entered "140" and the cursor is at the end of the input.

```
Bob:assignment2 Martin$ javac CalorieTracker.java
Bob:assignment2 Martin$ java CalorieTracker
Enter the calories in your starter:
140
```

Any single entry (**without spaces**) into the terminal while our program is running is considered to be a **token** of user input.

In this case, we want our program to **stop** and wait for a **single token of integer input**.

# USING A SCANNER (2)

We read single integer tokens using the following method in the Scanner class:

## **nextInt**

```
public int nextInt()
```

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in the default radix of this scanner.

### **Returns:**

the `int` scanned from the input

### **Throws:**

`InputMismatchException` - if the next token does not ma

```
in.nextInt();
```

*As this method gives us an integer value back, we can assign this value to a variable, or pass it around our program as we wish.*

# TRACKING A USER'S CALORIES (3)

```
import java.util.Scanner;

public class CalorieTracker {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        Person person = new Person();

        System.out.println("Enter the calories in your starter:");

        Dish starter = new Dish();
        starter.setCalories(in.nextInt())
    }
}
```

*Every time we invoke the nextInt method, our program will stop and wait for **another** token of user input.*

## TRACKING A USER'S CALORIES (4)

```
Scanner in = new Scanner(System.in);

Person person = new Person();

System.out.println("Enter the calories in your starter:");

Dish starter = new Dish();
starter.setCalories(in.nextInt());

System.out.println("Enter the calories in your main:");

Dish main = new Dish();
main.setCalories(in.nextInt());
```

*And we can do this as many times as we wish.*

## TRACKING A USER'S CALORIES (5)

```
System.out.println("Enter the calories in your desert:");

Dish desert = new Dish();
desert.setCalories(in.nextInt());

Meal meal = new Meal();
meal.setStarter(starter);
meal.setMain(main);
meal.setDesert(desert);

person.eat(meal);
System.out.println(person.getCalories());

in.close();
```

Compiling this code will result in a **warning** unless we call the **close** method on our Scanner instance.

## ASIDE: CLOSING SCANNER

We typically indicate when we have **finished** asking our user for input by **flagging** that our scanner can no longer be used for input using the **close** method.

- This also closes the **underlying resource** being used by the scanner.
- This avoids us **locking resources** (e.g. locking an input stream) but should be used carefully in this context, as input from the command line **cannot be re-opened**.
  - When dealing with user input, close is therefore usually designed to be the **last command** used within the program.
  - If it cannot be then this warning can be **ignored**.



## ASIDE: ENTERING DATA IN AN UNEXPECTED FORMAT

What would happen if I were to enter a **string** for the number of requested calories, rather than an integer?

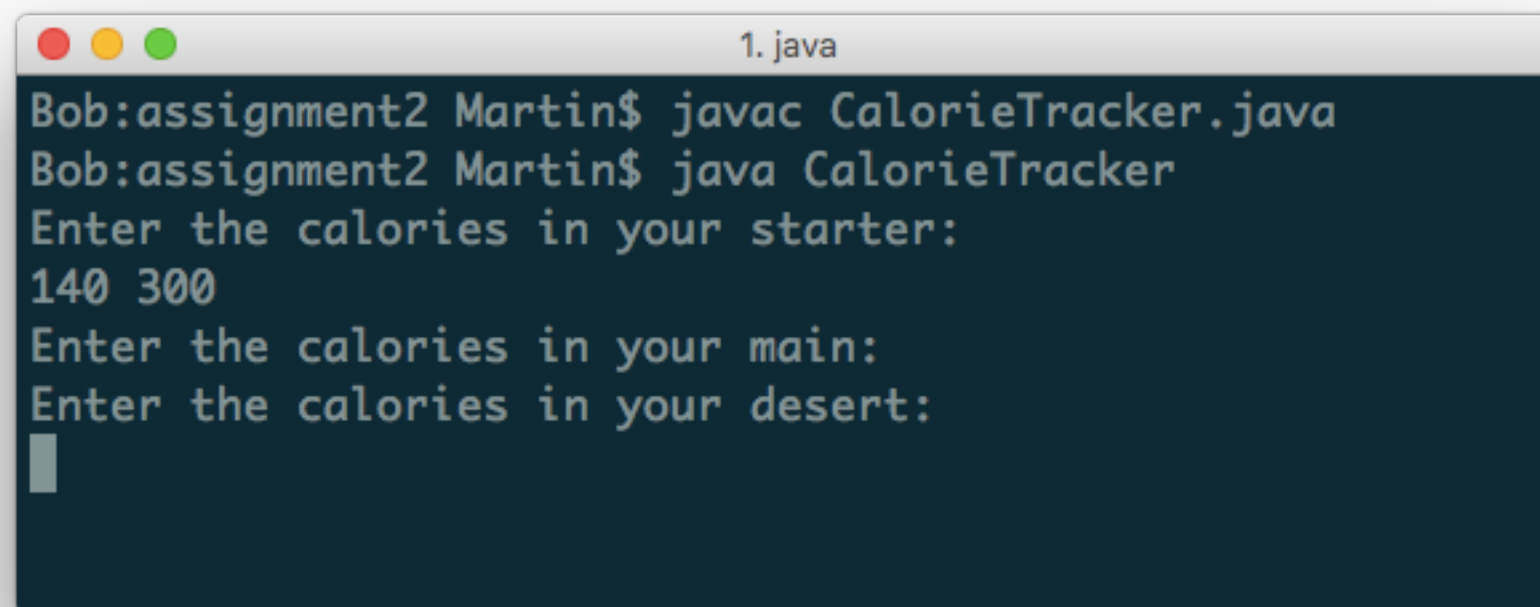
```
1. bash
Bob:assignment2 Martin$ javac CalorieTracker.java
Bob:assignment2 Martin$ java CalorieTracker
Enter the calories in your starter:
HELLO
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at CalorieTracker.main(CalorieTracker.java:17)
Bob:assignment2 Martin$
```

We'd get a **runtime error**. This is something we'll discuss later in the course.



## ASIDE: MULTIPLE TOKENS

Because our program currently deals with **tokens** as opposed to **lines** of input, if we decide to enter multiple tokens in one line, then we can receive unexpected behaviour:

A terminal window titled "1. java" with a dark background and light text. It shows the compilation and execution of a Java program. The user enters "140 300" as input for the "starter" section, demonstrating how multiple tokens on a single line are processed.

```
Bob:assignment2 Martin$ javac CalorieTracker.java
Bob:assignment2 Martin$ java CalorieTracker
Enter the calories in your starter:
140 300
Enter the calories in your main:
Enter the calories in your desert:
█
```

The program only pauses in expected places if there are **still tokens to read**, otherwise it will use the tokens that have already been entered.

# LECTURE EXERCISE: BACK TO THE BOOKS (1)

In the same way that we made our CalorieTracker more interactive, let's try and make our BookReader more interactive.

```
public class BookReader {  
  
    public static void main(String[] args) {  
  
        Book aGameOfThrones = new Book(704);  
        aGameOfThrones.readPage();  
        aGameOfThrones.readPage();  
        aGameOfThrones.readPage();  
        System.out.println(aGameOfThrones.readPageNumber());  
        aGameOfThrones.turnPageBack();  
        System.out.println(aGameOfThrones.readPageNumber());  
  
    }  
  
}
```

## LECTURE EXERCISE: BACK TO THE BOOKS (2)

Let's implement the following functionality:

- We have two books, 'A Game of Thrones' and 'A Clash of Kings', that constitute a (very small) **library**.
- We have the option to **read a book** from the library.
  - We are given the option to **select** one of these books.
  - When we do so, we read **10** pages from it.

# LECTURE EXERCISE: BACK TO THE BOOKS (3)

```
public class Library {  
  
    private Book aGameOfThrones;  
    private Book aClashOfKings;  
  
    public Library() {  
  
        aGameOfThrones = new Book(704);  
        aClashOfKings = new Book(768);  
  
    }  
}
```

# LECTURE EXERCISE: BACK TO THE BOOKS (4)

*We make one request for input and store the result.*

```
public void readABook() {
```

```
    AutomaticBookReader automaticBookReader = new AutomaticBookReader();
```

```
    Scanner in = new Scanner(System.in);
```

```
    String requestedBook = in.nextLine();
```

```
    if ( requestedBook.toLowerCase().equals("a game of thrones") ) {
```

```
        automaticBookReader.readBook(aGameOfThrones, 10);
```

```
    else if ( requestedBook.toLowerCase().equals("a clash of kings") ) {
```

```
        automaticBookReader.readBook(aClashOfKings, 10);
```

```
    else {
```

```
        System.out.println("Book not found.");
```

```
    }
```

```
    in.close();
```

```
}
```

```
}
```

*We built this class in Topic 4.*

**nextLine**

```
public String next
```

```
Advances this scanner
```

*Transforming the string to lowercase stops us being concerned about the capitalisation employed by the user.*



We can do some interesting things if we combine the use of a scanner with a loop.

In the laboratory, use the methods of the Scanner class to implement the following functionality:

- A program that keeps asking a user for numbers until they enter a '-1'. The total of these numbers should then be printed.
- A program that keeps asking a user for numbers until they enter a string. The total of these numbers should then be printed.

# A PREVIEW: DO WHILE

I mentioned in Topic 5 that there was another type of loop that I would talk about during Topic 6.

- This is because this loop is most commonly suited to user input:

*We get  
**one**  
**iteration**  
of the  
loop  
**before** the  
condition  
is tested.*

```
Scanner in = new Scanner(System.in);
```

```
String password; ← We could put a default value  
here (a sentinel value), but this  
reduces readability.
```

```
do {
```

```
    System.out.println("What's the password?");  
    password = in.nextLine();
```

```
} while ( !password.equals(this.password) );
```

- You might like to use a do while loop to help you with the previous questions.



# Arithmetic

---

# USING ARITHMETIC WHEN PROGRAMMING

We've already seen lots of examples of when arithmetic is useful when programming:

```
currentPage + 1;
```

```
balance - ( amount + 20 );
```

```
currentPage - 1;
```

```
balance - ( amount + 50 );
```

```
balance - amount
```

```
balance + amount
```

# ADDITION AND SUBTRACTION OPERATORS

Most of the **arithmetic operators** we've seen so far have been simple addition and subtraction operators.

The **form** of these operators intuitively **matches** their form in regular mathematics.

```
currentPage + 1;
```

```
currentPage - 1;
```

# MULTIPLICATION AND DIVISION OPERATORS

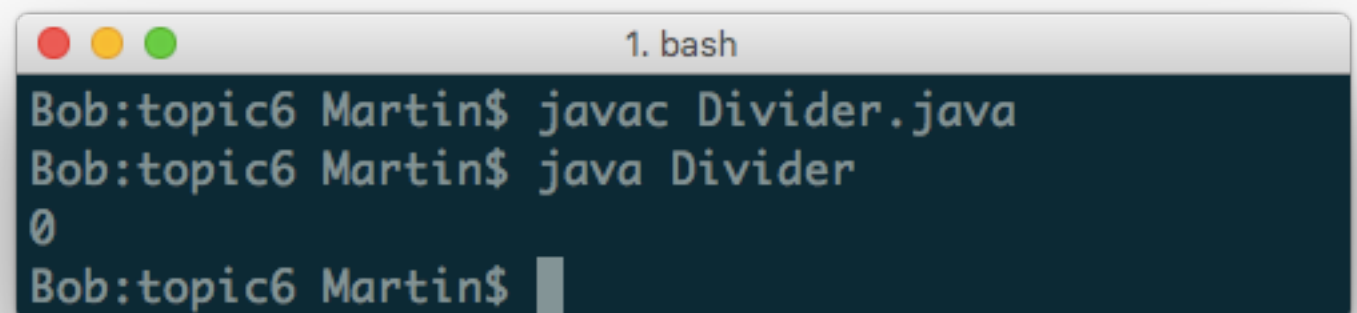
We have other operators available to us, but their formation differs from regular mathematics:

Java	Mathematical Notation	Description
+	+	Addition
-	-	Subtraction
/	÷	Division
*	×	Multiplication

# INTEGER DIVISION

When dividing, if at **least one** of the operands **isn't in floating point format**, the result of the operation **won't** have the required **precision**.

```
System.out.println(1/2);
```



```
1. bash
Bob:topic6 Martin$ javac Divider.java
Bob:topic6 Martin$ java Divider
0
Bob:topic6 Martin$
```

There are a number of simple solutions to avoid integer division:

```
System.out.println(1/2.0);
```

```
System.out.println(1.0/2);
```

```
System.out.println(1f/2);
```

```
System.out.println(1/2d);
```

## ASIDE: SHORT-CIRCUIT EVALUATION

Recall that operators of equivalent precedence are evaluated from left to right in almost all cases.

Because of this rule, programming languages like Java often employ a useful tool called **short-circuit evaluation**.

When the evaluation of a sequence of operators short circuits, it **cuts out** before the end. Languages like Java typically do this when the **truth value of a boolean operator can already be determined before completing the evaluation**.

We can leverage this behaviour in our programs, in order to ensure that conditions that would result in an error, given a state in the program, are not evaluated.

*check for dangerous thing && dangerous thing*

```
if ( b > 0 && 10 / b > 4 ) {
```

## ASIDE: TYPE RESULTING FROM AN ARITHMETIC OPERATION

The type resulting from an arithmetic operation is the same as the **operand with the highest precision**.

1/3.0

Double


1/3.0f

Float

1d \* 3l

Double

To confirm this, you can use the following boilerplate code:

```
System.out.println(((Object)()).getClass().getName());
```

Expression here

## ASIDE: FLOATING POINT ARITHMETIC ACCURACY (1)

A large number of decimal numbers have an **infinite representation in binary**.

- This is typically due to the **binary conversion process itself** (e.g. representing 0.1 in binary), or because those numbers themselves have an infinite decimal expansion (e.g.  $\pi$ ).

Representing these numbers in **finite memory** therefore often involves **rounding**, which results in **approximations** of true decimal values.

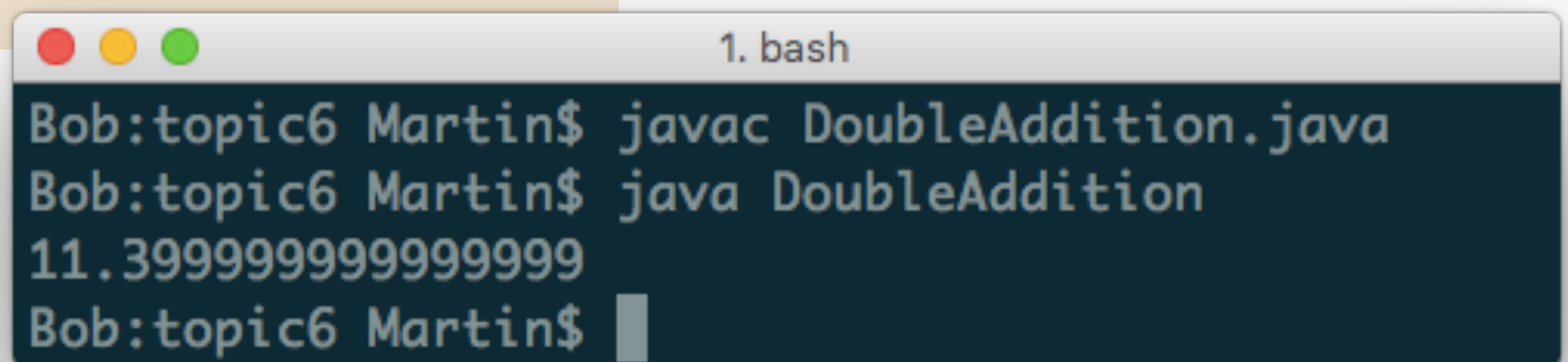
As such, the result of **arithmetic operations** with decimals may not always be accurate.



## ASIDE: FLOATING POINT ARITHMETIC ACCURACY (2)

For example, consider the following simple addition of double values:

```
System.out.println(5.6 + 5.8);
```



```
1. bash
Bob:topic6 Martin$ javac DoubleAddition.java
Bob:topic6 Martin$ java DoubleAddition
11.399999999999999
Bob:topic6 Martin$
```

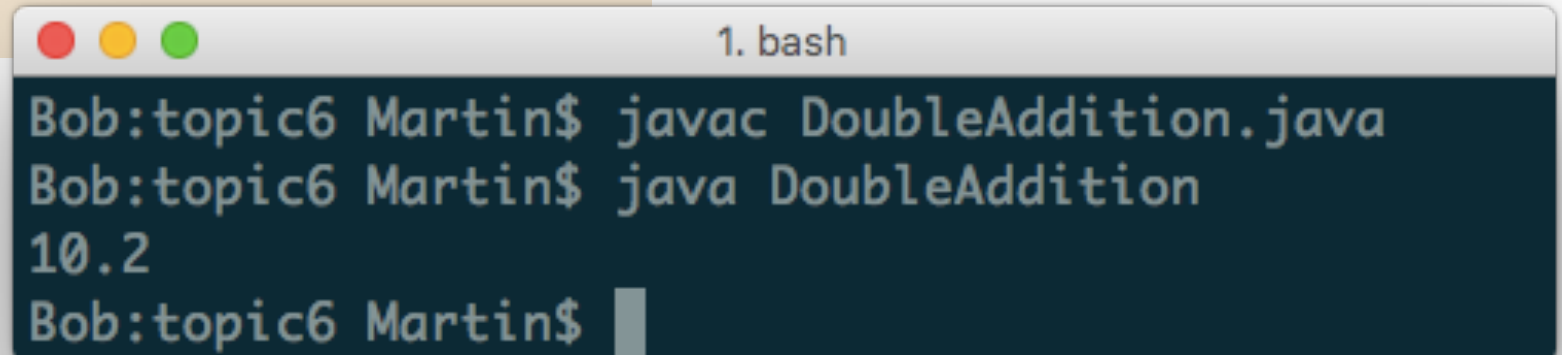
This occurs because, in reality, 5.6 and 5.8 (both of which have an infinite binary form), when converted to binary floating point, end up being rounded approximations of their true values.

- We can confirm this if we translate each of these values to (double precision) binary floating point and then convert them back to decimal again (receiving the (truncated) values 5.599999999999999999996 and 5.799999999999999999998, resp.).

## ASIDE: FLOATING POINT ARITHMETIC ACCURACY (3)

Even floating point arithmetic that appears to behave normal may actually just be the result of **fortunate rounding**.

```
System.out.println(5.1 + 5.1);
```

A terminal window titled "1. bash" with a dark background. It shows the execution of a Java program. The prompt is "Bob:topic6 Martin\$". The first command is "javac DoubleAddition.java". The second command is "java DoubleAddition", which outputs "10.2". The prompt returns to "Bob:topic6 Martin\$".

```
Bob:topic6 Martin$ javac DoubleAddition.java
Bob:topic6 Martin$ java DoubleAddition
10.2
Bob:topic6 Martin$
```

Here, a fuller precision result is actually 10.199999999999999999289..., but when rounded and stored as a double (typically to between **15 and 17** digits), the result given is 10.200000000000000000 (presented as just 10.2).

## ASIDE: FLOATING POINT ARITHMETIC ACCURACY (4)

One solution here, if exact results are required, is to use the **library class BigDecimal**.

```
BigDecimal fivePointSix = new BigDecimal("5.6");  
BigDecimal fivePointEight = new BigDecimal("5.8");  
System.out.println(fivePointSix.add(fivePointEight));
```

Other solutions include:

- Introducing a **tolerance** factor into if statements.
- **Rounding** (more shortly)
- Using **fixed point** representations (e.g. number of pennies)

But in this course we won't be **too** concerned with the small margins of error introduced by floating point numbers.

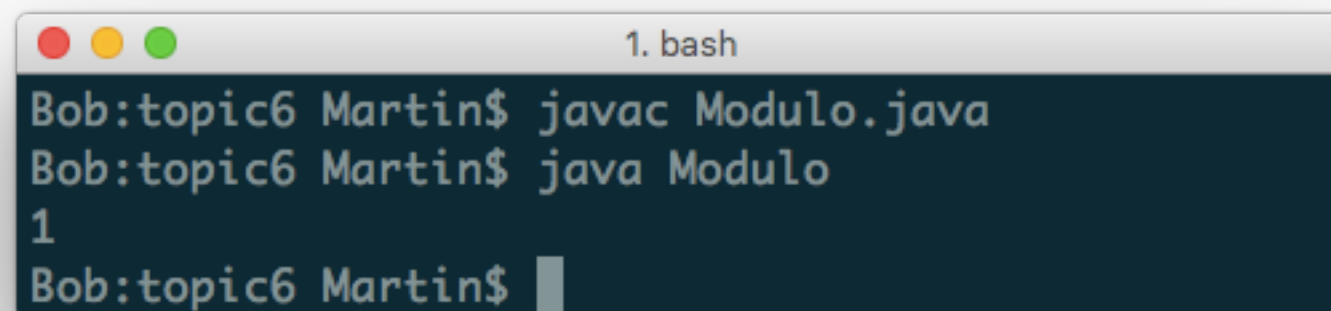


# THE MODULO OPERATOR

An operator that deserves a special mention is the **modulo** operator.

```
System.out.println(7 % 3);
```

As in regular mathematics, this returns the **remainder** after a division.

A terminal window titled "1. bash" with a dark background and light text. It shows the following commands and output:

```
Bob:topic6 Martin$ javac Modulo.java
Bob:topic6 Martin$ java Modulo
1
Bob:topic6 Martin$
```

Modulo allows us to do some interesting things, such as testing for **multiples**.

# LECTURE EXERCISE: FIZZ BUZZ (1)

It is commonly reported (<https://blog.codinghorror.com/why-cant-programmers-program/>) that many candidates for programming jobs during interviews **cannot solve the following problem:**

- ‘Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".'

# LECTURE EXERCISE: FIZZ BUZZ (2)

```
for ( int i = 1; i <= 100; i++ ) {  
    if ( i % 3 == 0 && i % 5 == 0 ) {  
        System.out.println("Fizzbuzz");  
    } else if ( i % 3 == 0 ) {  
        System.out.println("Fizz");  
    } else if ( i % 5 == 0 ) {  
        System.out.println("Buzz");  
    } else {  
        System.out.println(i);  
    }  
}
```

*There are lots of interesting optimisations to this basic solution, particular in terms of how multiples of both 3 and 5 are dealt with.*

# MORE POWERFUL MATHEMATICAL OPERATIONS (1)

Earlier we noted that **rounding** might be useful to help us with the (lack of) precision of floating point numbers.

There are also a number of other useful operations that aren't immediately representable using the basic arithmetic operations we've seen so far.

- What about raising 2 to the **power** of 8?
- What about finding the **square root** of 17?

Given the title of this topic, it probably intuitive that there is a library class available to help us with this.

## **Class Math**

```
java.lang.Object  
    java.lang.Math
```

---

```
public final class Math  
    extends Object
```

The class `Math` contains methods for performing basic numeric operation trigonometric functions.

Unlike some of the numeric methods of class `StrictMath`, all implemented the bit-for-bit same results. This relaxation permits better-performing im

By default many of the `Math` methods simply call the equivalent method in to use platform-specific native libraries or microprocessor instructions, w



# THE MATH CLASS

Methods from the `Math` class allow us to perform the operations noted previously:

```
Math.round((5.6 + 5.8) * 100.0) / 100.0;
```

*There are also other classes that allow us to round numbers.*

```
Math.pow(2, 8);
```

```
Math.sqrt(17);
```

*Detailed rules for these operations are given in the API.*

The `Math` class is also so regularly used that we do **not** need to import it.

# Subtopic: Static Methods And Fields

---

When we were using the methods in the `Math` class, you'll notice that we **didn't** make an object of the `Math` class before calling those methods.

Instead, we simply wrote the name of the class (`Math`) followed by the name of the method.

```
Math.pow(2, 8);
```

# STATIC METHODS (1)

As you've probably already seen from your own experiments, we can call a method in a class without creating an object of that class if that method is **static**.

```
static double          pow(double a, double b)  
                        Returns the value of the first argument raised
```

We've already seen several examples of static methods in the course, without explicitly identifying them as such:

```
System.currentTimeMillis();
```

```
System.nanoTime();
```

*We'll discuss the  
'System' part  
shortly.*

```
out.println();
```

## STATIC METHODS (2)

We can imagine a **single shared** copy of **all static identifiers**, existing **within** the **class type** itself, such that we can access those identifiers using our familiar **dot syntax**.

```
static double pow(double a, double b) {  
  
}  
  
static double sqrt(double a) {  
  
}
```



```
Math.pow(2, 8);
```

```
Math.sqrt(17);
```

But remember, as with our class copies, this is just a **simplified abstraction**, to help our understanding.

# BACK TO MARTIN PRINTER AND NUMBER PRINTER

Given this new (or not new) knowledge, we could, if we wanted to, go back and **evolve** some of early examples of methods, to instead be static.

- This **simplifies their use**, but leaving these methods non-static is still perfectly fine; a **style choice**.

```
public static void printMartin() {
```

```
MartinPrinter.printMartin();
```

```
System.out.println("+-----+");
```

```
System.out.print
```

```
System.out.print
```

```
}
```

```
public static void printNumber(int num) {
```

```
System.out.println("+-----+");
```

```
System.out.println("|" + num + "|");
```

```
System.out.println("+-----+");
```

```
NumberPrinter.printNumber(1);
```

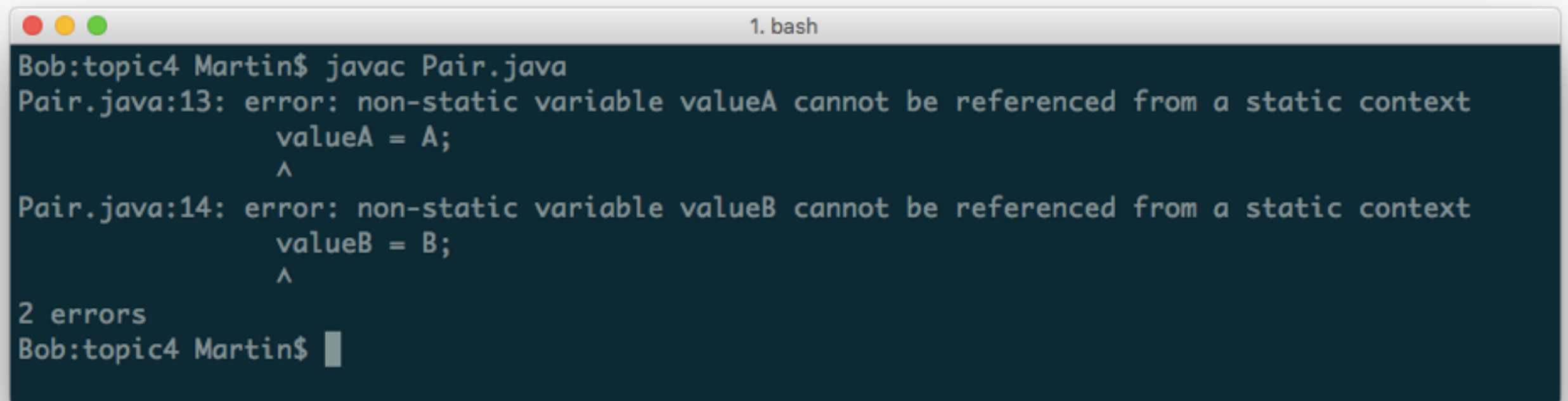
# THE CONSEQUENCES OF STATIC

This seems like quite a nice **shorthand**. Can we apply this to all our examples?

```
public class Pair {  
  
    private int valueA;  
    private int valueB;  
  
    public static void setPair(int passedValueA, int passedValueB) {  
  
        valueA = passedValueA;  
        valueB = passedValueB;  
  
    }  
  
    public static int getValueA() {  
        return valueA;  
    }  
  
    public static int getValueB() {  
        return valueB;  
    }  
}
```

Pair.setPair(1, 2);

# ERROR: NON-STATIC VARIABLE CANNOT BE REFERENCED FROM A STATIC CONTEXT

A terminal window titled "1. bash" with a dark blue background and white text. It shows the command "javac Pair.java" being executed. Two error messages are displayed: "Pair.java:13: error: non-static variable valueA cannot be referenced from a static context" with a caret pointing to "valueA", and "Pair.java:14: error: non-static variable valueB cannot be referenced from a static context" with a caret pointing to "valueB". The terminal concludes with "2 errors" and a new prompt line.

```
Bob:topic4 Martin$ javac Pair.java
Pair.java:13: error: non-static variable valueA cannot be referenced from a static context
    valueA = A;
    ^
Pair.java:14: error: non-static variable valueB cannot be referenced from a static context
    valueB = B;
    ^
2 errors
Bob:topic4 Martin$
```



## STATIC METHODS (3)

```
public static void setPair( ... ) {  
    valueA = passedValueA;  
    ...  
}  
  
public static int getValueA() {  
  
}  
  
public static int getValueB() {  
  
}
```

→ Pair.setPair(1, 2);

This error makes sense, because only the static methods are included in the shared copy of static identifiers, **not** the fields being referenced.

# PREVIEW: STATIC FIELDS (1)

```
public class Pair {  
  
    private static int valueA;  
    private static int valueB;  
  
    public static void setPair(int passedValueA, int passedValueB) {  
  
        valueA = passedValueA;  
        valueB = passedValueB;  
  
    }  
  
    public static int getValueA() {  
  
        return valueA;  
  
    }  
}
```

One solution would be to make these fields static too, and thus add them to the shared copy of static identifiers...

## PREVIEW: STATIC FIELDS (2)

```
private static int valueA;  
private static int valueB;  
  
public static void setPair( ... ) {  
    valueA = passedValueA;  
    ...  
}  
  
public static int getValueA() {  
  
}  
  
public static int getValueB() {
```

Pair.setPair(1, 2);

Pair.setPair(3, 4);

But hopefully it's clear that this would result in **undesired behaviour**, when we want to use **multiple** pairs.

- Within this **single** shared copy of static identifiers, we'd only be able to store **one set** of values at a time.

# STATIC VS. NON-STATIC METHODS

**What's the solution here?** Well, there isn't really one. Instead, static and non-static methods have different uses:

- Static methods are used when the operation within that method **does not involve non-static fields** (e.g. Number printer).
- We call these **instance fields (variables)**, because they are designed to hold **different values** for each instantiated object.
- Also, of course, if we wish to access static fields.
- Non-static methods are used to interact with and update instance fields, as we have seen (e.g. Pair).
- But they **can also** access static fields, **more to come**.



## ASIDE: UTILITY CLASSES

My main use for static methods is within **Utility** classes, that perform some common tasks that I will need multiple times:

```
63  /**
64   * A set of static utility methods.
65   *
66   * @author Martin
67   *
68   * public static double round(double value, int places) {
69   *     if (places < 0) throw new IllegalArgumentException();
70   *
71   *     BigDecimal bd = new BigDecimal(value);
72   *     bd = bd.setScale(places, RoundingMode.HALF_UP);
73   *     return bd.doubleValue();
74   * }
```

*This class would have a private constructor.*

*Every variable referenced here is defined within the scope of the method.*

It's worth saying that it is possible to call **static methods** and **fields** via an object.

But this is **conceptually awkward**, and thus you are **warned** by the compiler when doing this.

So why permit this?

- An error in the **design** of the language?
- Most languages are **not** perfect.

# REMEMBER: BACK TO MAIN

```
public static void main(String[] args) {
```

We now know a **even more** about **main**.

- We know that main can **never return** anything (where would it go anyway?)
- This is a **parameter**, so main must **accept** some information.
- But unfortunately, we **still** aren't in a position to discuss what is passed to the main method.
- We know that main has to be **visible** from outside the class, so that it can be run by the JVM (Topic 4).
- We know that main can be accessed without having to make an object of our driver class, which makes things **easier** for the JVM.

There was an **implication** before that static fields were somehow **bad** because they can't be **virtually** copied in order to store **different values**.

- This contrast instance fields, which are virtually copied with each object of a class, and thus allow us to store different values.

But static fields do have a place, both **conceptually** and **practically**.



# REMEMBER: CLASSES AND OBJECTS IN THE REAL WORLD (3): STATE

The fields of these classes are clear; **common features that we can all identify as a part of the shared concept.**

But fields often hold different data; things in the world are in different **states.**



# STATIC FIELDS IN THE REAL WORLD

```
public class Dog {  
    private String colour;  
    private int earLength;  
    private static int numberOfLegs;  
    private static int numberOfEyes;  
}
```

It's true that different instances of our dog will have different values for attributes such as these.

But there are also attributes that **will be the same for each instance.**

It therefore makes sense for these fields to be **static**.

*We say that these fields are **part of the concept** (the **class**; the **shared static copy**), rather than the **instance**.*

*A change in the concept, should invoke a change in all the instances.*



# FRIENDS OF STATIC (1): FINAL

When a field is static because it is **being used to** represent an attribute of a class (concept) that **is the same** for each instance, it doesn't really make conceptual sense to make this field **variable**.

- A variable field implies that this attribute will be used to hold multiple different values, which it won't in this case.

So, instead, we **should** mark this field as a **constant**, which is the **opposite of a variable**.

```
private static final int numberOfLegs;  
private static final int numberOfEyes;
```

We do this by using the modifier **final**.

## FRIENDS OF STATIC (2): FINAL

When we make a field constant instead of variable, there are a couple of **other** things we should also do:

- We should **assign values** to that field immediately.
- I noted priorly that this was bad practice. This is the **only exception** to this rule (in my opinion).
- Because the value will never be changed, there is no chance of inefficient reassignment.
- This is also the **only** place we can assign the field (non-static final fields can also be assigned in the constructor).

```
private static final int numberOfLegs = 4;  
private static final int numberOfEyes = 2;
```

When we make a field constant instead of variable, there are a couple of things we should also do:

- We should write the variable name in **uppercase**
- This signifies that the variable is a constant.

```
private final static int NUMBER_OF_LEGS = 4;  
private final static int NUMBER_OF_EYES = 2;
```

## FRIENDS OF STATIC (4): PUBLIC

When we make a field constant instead of variable, there are a couple of things we should also do:

- Finally, we should also make this variable **public**.
  - Again, I noted priorly that this was bad practice, but this should be the **only** exception to this rule (in my opinion).
  - This doesn't violate **encapsulation**, because the value can never be changed.
  - This allows us to **efficiently view** the data in the field, to quickly **learn** about the features of a class (without static this would not be possible).

```
public final static int NUMBER_OF_LEGS = 4;  
public final static int NUMBER_OF_EYES = 2;
```

# PRACTICAL USES OF PUBLIC CONSTANT STATIC FIELDS

While public, constant, static fields have a **conceptual** relevance, they also have a **practical** use (non-exhaustive):

- They can allow us to quickly extract important, practical information from a class.

```
/**
 *
 */
public final static String MARTINPREFIX = "/Users/Martin/Dropbox/workspace/SearchGames/";
```

- They can allow us to quickly extract important **objects** from a class.

System.**in**

System.**out**.println("");

## Field Detail

### in

```
public static final InputStream in
```

The "standard" input stream. This stream is already open by the host environment or user.

### out

```
public static final PrintStream out
```

The "standard" output stream. This stream is already open by the host environment or user.

## ASIDE: ENUM VS. PUBLIC STATIC FINAL (1)

Another common use for public statics final fields is to define our own **keywords**, which can help improve the readability of our code:

```
public class BankAccountTypes {  
    public static final int CURRENT = 0;  
    public static final int SAVINGS = 1;
```

```
public BankAccount(int type) {  
    if ( type == BankAccountType.CURRENT ) {
```

```
public static void main(String[] args) {
```

```
    BankAccount account = new BankAccount(BankAccountType.CURRENT);
```



## ASIDE: ENUM VS. PUBLIC STATIC FINAL (2)

Java provides **enum** types to support this practice more **formally**.

- More specialised, but have to be defined as a separate type.

```
public enum BankAccountType {  
  
    CURRENT, SAVINGS;  
  
}
```

```
public BankAccount(BankAccountType type) {  
  
    if ( type == BankAccountType.CURRENT ) {  
  
        ...  
  
    }  
  
}
```

```
public static void main(String[] args) {  
  
    BankAccount account = new BankAccount(BankAccountType.CURRENT);  
  
}
```



# NON-CONSTANT, NON-PUBLIC STATIC FIELDS (1)

Previously we saw that non-constant, non-public static fields were typically undesirable, as they only exist within the shared copy of **static identifiers**, and thus cannot be used to store different values for each object.

- We can see this issue more explicitly if we access these static fields through an object.

```
private static int valueA;  
private static int valueB;
```

```
public void setPair(int valueA, int valueB) {  
    this.valueA = valueA;  
}
```

```
Pair pairA = new Pair();  
pairA.setPair(1, 2);
```

*We should really qualify valueA here with the class name. **Open question:** Why is this possible, given valueA is private?*

## NON-CONSTANT, NON-PUBLIC STATIC FIELDS (2)

Non-constant (and thus non-public) static fields also have a use.

- Why is it useful to update the values in a field in **all objects** of a class?
- A common reason is to **record the number of instances that are created of a class.**

```
private static int numberOfPairs;  
  
public Pair() {  
    numberOfPairs++;  
}
```

```
public static int getNumberOfPairs() {  
    return numberOfPairs;  
}
```

**Why is this useful?**

# LECTURE EXERCISE: BLOCKCHAIN (1)

Blockchain technology (distributed ledger technology) gives us a secure, ordered and replicated way to store data (typically financial data).



Let's represent a Miner who can add Blocks to a Blockchain.

- We'll do this in a very **abstract way**.

# LECTURE EXERCISE: BLOCKCHAIN (2)

When representing blocks for a blockchain:

- There must be a sequential order to the blocks.
- We must know how many blocks there are.
- We cannot hold any information about the sequencing of the blocks centrally (i.e. the information must be held in the blocks themselves).
- It should not be possible for a user to manipulate this ordering.

## Timestamp?

- If multiple blocks are generated at the same time, a timestamp will be useless.

# LECTURE EXERCISE: BLOCKCHAIN (3) - BLOCK

```
public class Block {  
  
    private static int blocks;  
    private int blockNumber;  
  
    public Block() {  
        blockNumber = blocks++;  
    }  
  
    public int blockNumber() {  
        return blockNumber;  
    }  
}
```

*Here we use the information in a static field to give each block a unique, sequential number.*

*Keeping both these values **private** is important for the integrity of our numbering system.*

# LECTURE EXERCISE: BLOCKCHAIN (4) - MINER AND BLOCKCHAIN

```
public class Miner {  
    public static void main(String[] args) {  
        Block myBlock = new Block();  
        Blockchain chain = new Blockchain();  
        chain.addBlock(myBlock).  
    }  
}
```

How do we do this?  
**On to Topic 7!**

```
public class Blockchain {  
    Keep a list of blocks  
    public void addBlock(Block block) {  
        Add the block to a list  
    }  
}
```

# Final Thoughts

---



With so many libraries available to us, a natural question is **can't we use a library for everything?**

- When working as developers, we should try and reuse code via libraries as much as possible.
  - Or at least tailor the functionality of an existing library.
- When completing assignments during our education, it's best to only use a library if that library has been covered already during lectures, as the functionality you're reusing may be the subject of assessment.
  - The coursework will only require libraries that have been shown.

## ASIDE: CONFLICT BETWEEN USER CLASSES AND LIBRARY CLASSES

Be aware of the existence of library classes when defining your own class names, as this can cause **unexpected behaviour**.

```
public class String {
```

}

```
public static void main(String[] args) {
```

```
String x = "abc";
```

## 1. bash

```
Bob:topic6 Martin$ javac InnocentDriver.java
```

```
InnocentDriver.java:5: error: incompatible types: java.lang.String cannot be converted to String
```

```
String x = "abc";
```

^

1 error

```
Bob:topic6 Martin$
```

## ASIDE: MORE LIBRARY CLASSES TO EXPLORE

We will see lots more examples of library classes in this course, especially next semester, when we look at library classes for creating **graphical user interfaces**.

In the interim, you can look at the library classes for the following pieces of functionality:

- Generating random numbers
- Time
- File Input

# Topic 6: Library Classes

Programming Practice and Applications (4CCS1PPA)

---

Dr. Martin Chapman  
Thursday 27th October

[programming@kcl.ac.uk](mailto:programming@kcl.ac.uk)  
[martinchapman.co.uk/teaching](http://martinchapman.co.uk/teaching)

**These slides will be available on KEATS, but will be subject to ongoing amendments. Therefore, please always download a new version of these slides when approaching an assessed piece of work, or when preparing for a written assessment.**