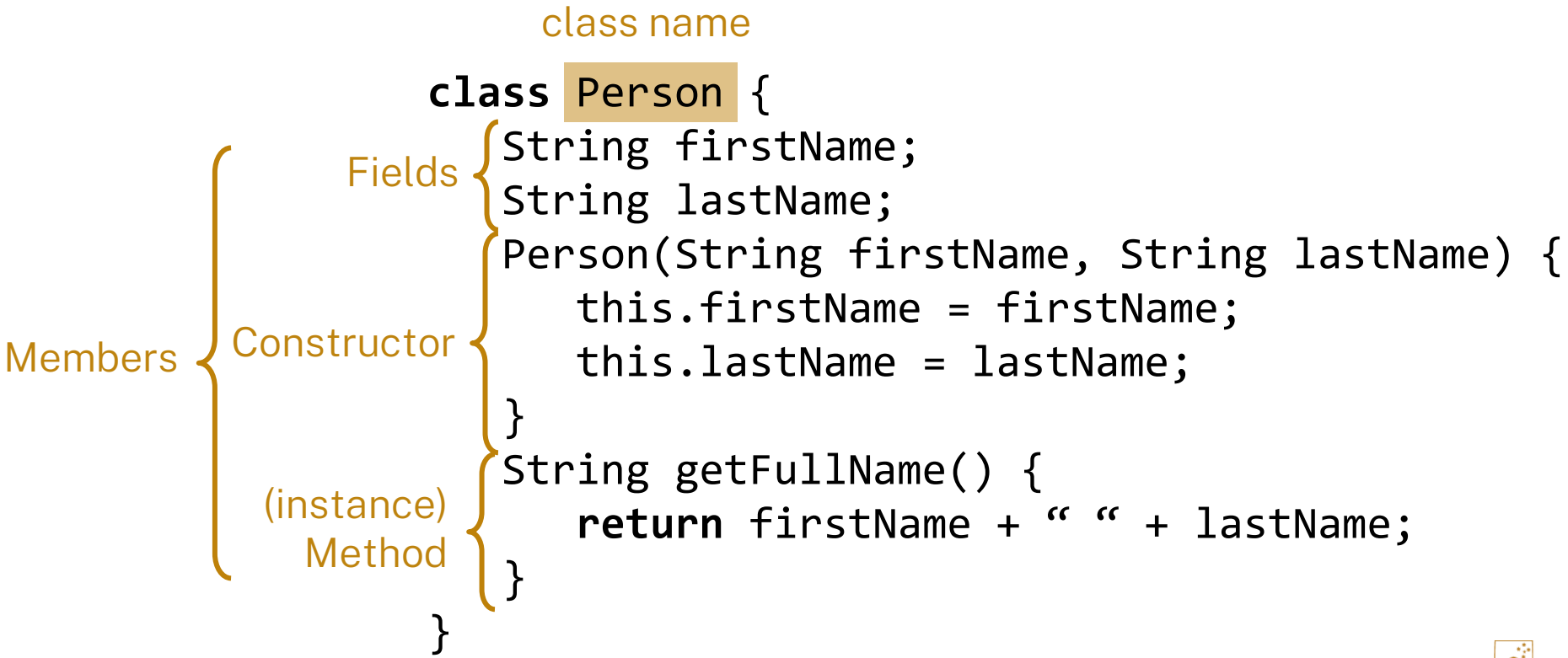
Structured Programming

COMP1110/6710

Image Courtesy NASA/JPL-Caltech.

Recap + Definitions: Classes



More on Methods

Every reference value has methods

Australian National University

Any reference value

Things we'll cover in this course:

```
"hello".equals("world")
    ⇔ Equals("hello", "world")

new HashMap<>().toString()
    ⇔ ToString(new HashMap<>())

CurrentDate().getHashCode()
```

Things we likely won't cover:

```
"hello".getClass()
"hello".notify()
"hello".notifyAll()
"hello".wait()
```


Strings

Functional Java:

```
Concatenate(“hello”, “world”)
SubString(“hello”, 2, 4)
Contains(“hello”, “ell”)
Replace(“ell”, “ih”, “hello”)
Length(“hello”)
GetCharAt(“hello”, 1)
…
```

Java:

```
“hello”.concat(“world”)
“hello”.substring(2, 4)
“hello”.contains(“ell”)
“hello”.replace(“ell”, “ih”)
“hello”.length()
“hello”.charAt(1)
…
```

See also https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/lang/String.html

Lambdas/Function References

Remember `function.apply(…)` ? `predicate.test(…)` ?

Turns out those are method calls on function objects.

Method Calls

[expression] . [method-name] ([expression] …)

“receiver” other arguments

becomes “this” in method code

For an “instance” method call, you always need a receiver.

What if you don’t have one?

Static Members

Class members of which there is only one copy

- No different values for different objects

Australian National University

Static Fields

Access from outside the class:
`Bean.counter`

In general:
`[class name] . [static field name]`

```
class Bean {
    int number;
    static int counter;
    Bean() {
        number = counter++;
    }
}
```

A single field shared by all `Bean`s

Seen with for-loops before. Increases variable by 1, returns old value from before increase.

Static Methods

Essentially global functions, but grouped into a class

Functional Java:

```
StringToInt(“24”)
StringToDouble(“5.2”)
…
Sin(0.5)
Round(0.5)
RoundInt(0.5)
…
```

Java:

```
Integer.parseInt(“24”)
Double.parseDouble(“5.2”)
…
Math.sin(0.5)
Math.round(0.5)
Math.toIntExact(Math.round(0.5))
…
```

Public Static Void Main

Or: main methods in Java

```
[import statements go here]

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

In a file called "MyClass.java"

No "this" available inside a static method!

Public Static Void Main

Or: main methods in Java

```
[import statements go here]

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

In a file called "MyClass.java"

A class

Public Static Void Main

Or: main methods in Java

```
[import statements go here]

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

In a file called "MyClass.java"

A static field access (to a `PrintStream`)

Public Static Void Main

Or: main methods in Java

```
[import statements go here]

public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

In a file called "MyClass.java"

An instance method call

Wait – how did this work before?

Distinction-Level Content

`--enable-preview` does several things:

- Wrap everything in a class named like the file
- Enable instance main methods (so main does not have to be static)
- Enable main methods without `String[] args`
- Automatically import a number of standard libraries
- Add some special standard libraries for `println`, `print`, and `readln`

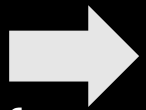
FYI: https://openjdk.org/jeps/477

Wait – how did this work before?

Distinction-Level Content

If file is called “Test.java”

```
void main() {
    println(sum(1, 2));
}
int sum(int a, int b) {
    return a + b;
}
```



```
import static java.io.IO.*;
public class Test {
    public static void main(String[] args) {
        new Test().main();
    }
    void main() {
        println(sum(1, 2));
    }
    int sum(int a, int b) {
        return a + b;
    }
}
```

FYI: https://openjdk.org/jeps/477

The Design Recipe

Adjustments for Classes

Australian National University

1. Problem Analysis and Data Design

For the purposes of this step, for now, classes are essentially records, with the following adjustments:

- @param annotations go the the constructor(s)
- Each field will still need an interpretation on its own.

The template is like a record template, i.e. essentially a list of fields. As with records, this may become more interesting with recursion.

1. Problem Analysis and Data Design

```
/** Represents a monotonically increasing counter.
 *  Examples: Counter(5), Counter(10532)
 *  @implSpec Invariant: the counter value only ever
 *  increases.
 */
class Counter {
  /** The current value of the counter >= 0 */
  int counterValue;
  /**
   * Creates a new Counter
   * @param startValue – the starting value of the counter >= 0
   */
  Counter(int startValue) {
    counterValue = startValue;
  }
}
```

2. Purpose Statement & Signature

Same as for functions, for all of

- (instance) methods
- Static methods
- Constructors

Note: constructors don’t have an @return spec.

Particularly important: effects, pre/postconditions, invariants

3. Examples

Same as for functions, for all of

- (instance) methods
- Static methods
- Constructors

Note: constructors that just assign fields directly from arguments don’t need examples.

4. Design Strategy

Same as before, but you may now nest case distinctions (to a reasonable degree. Don’t make your code too complicated – create helper functions!).

You also do not have to include return statements in branches. You may continue after an if-statement, and omit the else-branch.

Finally, a new Design Strategy: iteration

4. Design Strategy: Iteration

Similar to Case Distinction, but more important. If you are doing both, call the Design Strategy “Iteration”.

Iteration allows you to use loops (for/enhanced for/while/do-while).

Iteration may be nested (again, to a reasonable degree), and multiple loops can follow each other (yet again, to a reasonable degree).

IMPORTANT: each loop needs a comment on why it should terminate (where applicable)

4. Design Strategy: Iteration

```
// i is not assigned within body,
// but increased towards end
// condition at every iteration
for(int i=0; i<10; i++) {
    ...
}
```

```
// iterating over finite-size data
// structure
for(String name : names) {
    ...
}
```


4. Design Strategy: Iteration

```
// termination upon user input
while(true) {
    if(readln().equals("q")) {
        break;
    }
}
```

```
// integer bounds are always
// moving closer together
do {
    ...
    if(...) {
        i++;
    } else {
        j--;
    }
} while(i < j);
```


5. Implementation

Essentially unchanged, modulo relaxations in design strategies, and new language features.

6. Tests

Still need to write tests, and turn examples into tests.

- JUnit [Demo, see also https://www.jetbrains.com/help/idea/junit.html#intellij; use org.junit.jupiter:junit-jupiter:5.9.0 or higher]

Files

Input/Output – Part 3

Making things actually useful

Australian
National
University

Many things in computing are about the right files with the right content being in the right place.

Exploring the File System

java.io.File

```
File file = new File("myFile.txt");
```

A text file in the folder where your program is executed

```
File file = new File(".");
```

The folder where your program is executed

```
File file = new File("../parentFile.bla");
```

Some file in the folder above where your program is executed

java.io.File

Short for “if and only if”

file.exists() – a boolean, true iff the file/directory exists

file.isDirectory() – a boolean, true iff the path specifies a directory

file.isFile() – a boolean, true iff the path specifies a “normal” file

file.getParentFile() – a File, representing the parent directory of the file

file.listFiles() – a File[], representing all the files contained in a directory

file.mkdir() – creates a directory at the path represented by the file

file.mkdirs() – like mkdir, but also creates all necessary parent dirs

Reading (text) Files

```
try(var reader = new BufferedReader(new FileReader(file))) {
   for(String line = reader.readLine(); line != null;
               line = reader.readLine()) {
       … [do something with line] …
   }
} catch(Exception e) {
  throw new RuntimeException(e);
}
```

readLine returns null when the reader has reached the end of a file.
Since files have finite size, this terminates.

Java's mechanism for error handling forces you to handle some potential errors (warnings will say something about "unhandled exceptions").
For now, this code just says that in those cases, we want to crash the program.

Writing (text) Files

Putting a reader/writer in such a block ensures that the file is closed when you are done with it or an error occurs. That's very useful!

```
try(var writer = new BufferedWriter(new FileWriter(file))) {
    … writer.write("Hello"); …
    writer.newLine(); …
    writer.write("World\n");
} catch(Exception e) {
  throw new RuntimeException(e);
}
```

Two alternative ways of adding newlines to your text. Stick to one of them.

`writer.newLine()` may adjust to your Operating System. On Windows, newlines are traditionally "\r\n", though it can also deal with "\n".

Practice (at home)

Write a program for yourself that helps you keep your notebook.yml .

For example, it could work such that you can run it with:

```
java Notebook start “Part 2” “implementing tree for part 2”
```

at the start of your session, and with:

```
java Notebook end
```

at the end of your session. The program would automatically add the relevant session timestamps, comment, and add the corresponding minutes to the “Part 2” entry.