COMP1110/6710

Structured Programming

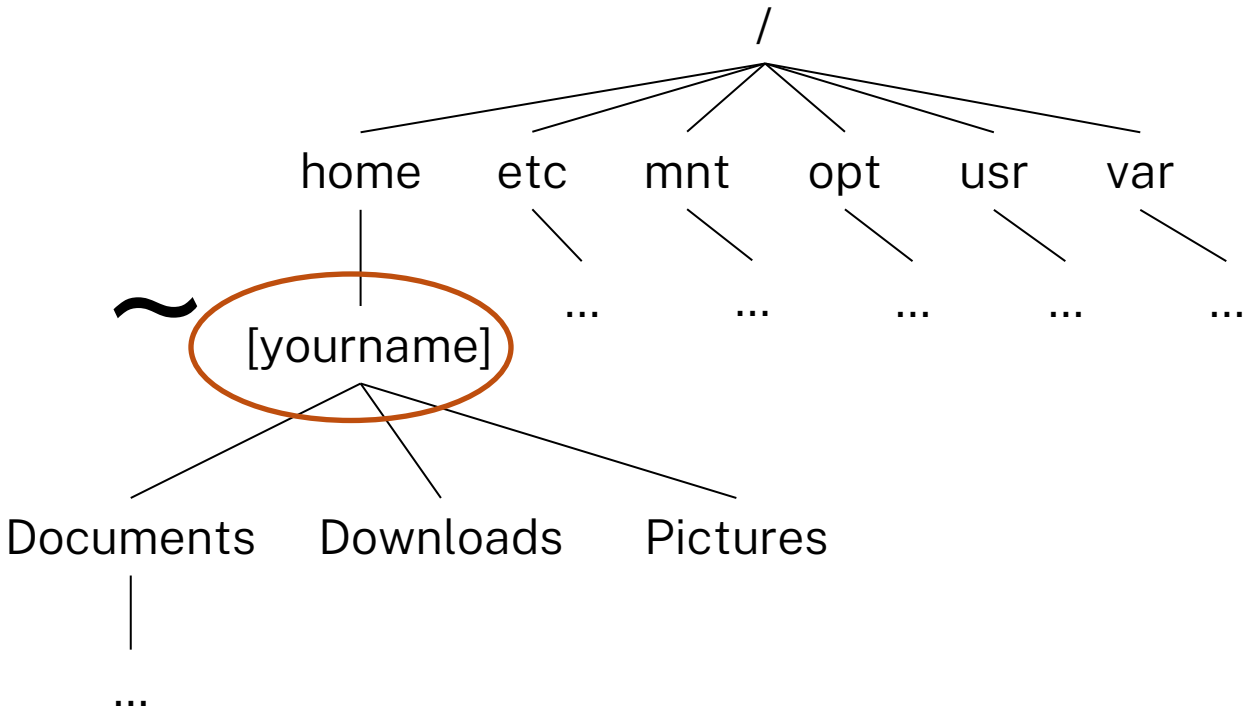
Australian National University

pollev.com/fabianm
Register for Engagement

Needs ANU Account!

Image Courtesy NASA/JPL-Caltech.

File Systems and Trees



Git

[Live Demo]

See https://comp.anu.edu.au/courses/comp1110/notes/git-and-notebooks/

And https://missing.csail.mit.edu/2020/version-control/

Functional Java

Follow along on your laptops!

Australian National University

Previously in COMP1110

```
void main() {
    println(“Hello World”);
}
```

Function Body = Code

Function Signatures

Function Signature

```
void main() {
    println(“Hello World”);
}
```

Function Signatures

Return Type *Name* *Arguments*

```
void main() {
    println("Hello World");
}
```

`main` is a special function – it's where the program starts running

`void` is a special type – it is the return type of functions that do not return anything. We only use it for special functions.

Function Signatures

Return Type *Name* *Arguments*

```
void main(String... args) {
    println("Hello World");
}
```

`main` is the standard entry point of all programs – it is where the program starts running

`void` is a special type – it is the return type of functions that do not return anything. We only use it for special functions.

Using a Program Argument

```
void main(String... args) {
    println(“Hello ” + args[0]);
}
```

Array Accessor

… and Array Accessors are special constructs for now.
You should only use them in the main function, if at all.

Statements

```
void main(String... args) {
    println(“Hello ” + args[0]);
}
```

What are these for?

Statements

```
void main(String... args) {
    println("Hello " + args[0]);
    println("How are you?");
}
```

Statements are evaluated top-to-bottom

More Statements!

Braces {} enclose *Blocks* of *Statements*.
Each *Statement* is terminated by a ;

More Functions!

```
String addBang(String str) {
    return str + "!";
}

void main(String... args) {
    println("Hello " + args[0]);
}
```

Not void!

A return statement

Return statements have to be the last in a block
Non-void methods need to return something

Expressions

```
String addBang(String str) {
    return str + “!”;
}

void main(String... args) {
    println(“Hello ” + args[0]);
}
```

An expression

Expressions produce values
Statements do not
Both may consume values

Expressions

```
String addBang(String str) {
    return str + “!”;
}

void main(String... args) {
    println(addBang(args[0]));
}
```

Another function call

Expressions

```
String addBang(String str) {
    println(str);
    return str + “!”;
}
void main(String... args) {
    println(addBang(addBang(args[0])));
}
```

Nested expressions,
Like function calls,
Are evaluated inside-out

And Another function call

Expressions

```
String addBang(String str) {
    println(str);
    return str + “!”;
}
void main(String... args) {
  println(addBang(args[0])+addBang(args[1]));
}
```

Sibling expressions are evaluated left to right

Two function calls on the same level

Importing Stuff

Download Standard Library from
https://comp.anu.edu.au/courses/comp1110/notes/functional-java/#the-standard-library

```
import static comp1110.lib.Functions.*;
```

Import a particular set of functions

```
void main(String... args) {
  println(StringToInt(args[0]) + StringToInt(args[1]));
}
```

A newly available function

Basic Data Types

Type	Represents	Examples
String	Text	“” ; “Hello World!”
boolean	True/False	**`true`** ; **`false`**
int, long	Integer numbers	0 ; 1 ; 2 ; 15; 42 ; 123534; 15l
float, **double**	Floating-Point numbers	0.15 ; 1.0 ; 2352.634 ; 14.0f
char	Characters (Unicode)	‘a’ ; ‘’

Do not use this where Precision is important, e.g. money!

Control Flow – `if` Statement

Statements are evaluated top-to-bottom

```
void main() {
    …
    if(someBoolean) {
        …
    } else {
        …
    }
    …
}
```

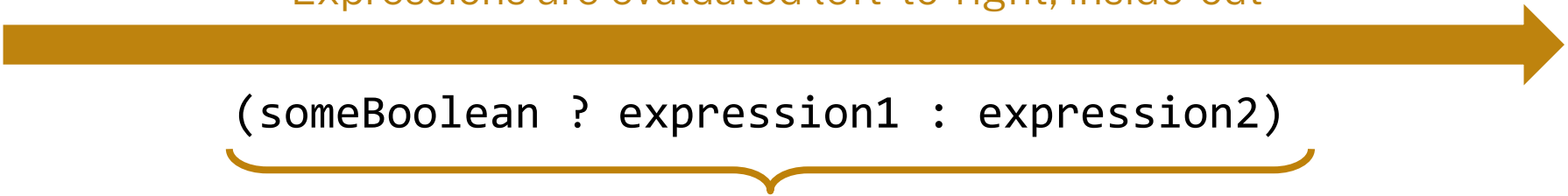
Evaluate these if `someBoolean` is `true`

Evaluate these if `someBoolean` is `false`

For Functional Java, each branch needs **`return`** statement, and no other statements may follow

Control Flow – `if` Expression

Expressions are evaluated left-to-right, inside-out



```
(someBoolean ? expression1 : expression2)
```

Produce value of `expression1` if `someBoolean` is `true`,
else Value of `expression2`
In both cases, the other expression is not evaluated

Control Flow – switch Statement

Statements are evaluated top-to-bottom

```
void main() {
    …
    switch(someInteger) {
        case 0: …; return …;
        …
        case 326: …; return …;
        default: …; return …;
    }
}
```

Can be any basic types and some others as long as cases match

Evaluate these if `someInteger` is `0`

…

Evaluate these if none of the above match

Control Flow – switch Expression

Statements are evaluated top-to-bottom

```
void main() {
    …
    return … switch(someInteger) {
        case 0 -> …;
        …
        case 326 -> …;
        default -> …;
    } … ;
}
```

Can be any basic types and some others as long as cases match

Evaluate these if `someInteger` is `0`

…

Evaluate these if none of the above match

Constants & Intermediate Values

```
int MAX_SPEED = 110;
void main(String... args) {
    int speed = StringToInt(args[0]);
    if(speed > MAX_SPEED) {
        println(MAX_SPEED);
    } else {
        println(speed);
    }
}
```

Global Constant Definition

Intermediate Value Definition
Compare with let-expression in Haskell

Basic Operations

Operator	For	Operation
+	String, numbers	append or plus
- ; * ; / ; %	numbers	Minus ; times ; divide ; remainder
! ; && ; ||	booleans	not ; and ; or

The standard library gives you functions like Equals, GreaterThan, LessThan .
For numbers and Booleans, you can also write == ; > ; <

The Design Recipe

Step 1. Problem Analysis and Data Design

Australian National University

Basic Data Types

Type	Represents	Examples
`String`	Text	“” ; “Hello World!”
`boolean`	True/False	**`true`** ; **`false`**
`int`, `long`	Integer numbers	0 ; 1 ; 2 ; 15; 42 ; 123534; 15l
`float`, **`double`**	Floating-Point numbers	0.15 ; 1.0 ; 2352.634 ; 14.0f
`char`	Characters (Unicode)	‘a’ ; ‘🐯’

The standard library adds opaque types: `Date`, `DateTime`, `Colour`, `Image`

Enumerations

For when you have a limited number of distinct cases.

```
/** A TrafficLightState indicates whether traffic can flow in
  * a particular direction at a given point in time.
  */
enum TrafficLightState {
    /** Traffic must stop */
    RED,
    /** Traffic must stop if able */
    YELLOW,
    /** Traffic can go ahead */
    GREEN
}
```

Need to have a comment
that explains the data definition overall,
as well as comment for every case.

This definition enables you to write TrafficLightState.RED, TrafficLightState.YELLOW, and TrafficLightState.GREEN in any expression location.

Templates

Every explicit data definition comes with a template

For enumerations, the general scheme is:

```
// { …
//   return  … switch(x) {
//     case [case1] -> …;
//     […];
//     case [caseN] -> …;
//   } …;
// }
```

Key Motto:

The shape of the data determines the shape of the code!

Template Example: TrafficLightState

Every explicit data definition comes with a template

For the definition of TrafficLightState from earlier, the template is:

```
// { …
//   return  … switch(tls) {
//     case RED -> …;
//     case YELLOW -> …;
//     case GREEN -> …;
//   } …;
// }
```

Exercise - what's the template for:

```
/** A CourseResult represents a student's outcome of taking
  * a course on a coarse level */
enum CourseResult {
    /** Passed the course with one of HD, D, CR, P, PS, CRS */
    PASS,
    /** Failed the course with one of N, NCN, CRN, WN */
    FAIL,
    /** Still awaiting a result with one of DA, KU, PX, RC, RP,
      * WA, WF, or no grade yet */
    PENDING,
    /** Withdrew from the course without failure through WD, WL */
    WITHDRAWN,
    /** Some other special kind of result (e.g. EE) */
    OTHER
}
```

Records

For when data consists of more than a single piece of information.

Overall purpose of data definition

```
/** A Student is a record that contains key information about an
  * ANU student. Examples:
  * - Lisa Studywoman, u1234567, BAC
  * - Paul Masterson, u7654321, MCOMP
  * @param name – the name of the student, a non-empty String
  * @param uid – a UID, identifying the student
  * @param program – the student's degree program
  */
record Student(String name, String uid, String program) {}
```

Example values

Explanations of fields

javadoc format, useful for later

The fields of the record

Record Templates – a bit boring

Every explicit data definition comes with a template

For records on their own, the general scheme is:

```
// { … x.f1() … […] … x.fN() … }
```

This assumes a record X with N fields of names f1…fN .

All we are saying here is that code may access the fields of a record.

Note: templates to not restrict the order or number of such field accesses. You can arbitrarily copy, move, and remove them.

Records – Creating and Using

To create a record, use the expression form:

```
new RecordName(valueForField1,…,valueForFieldN)
```

For example:

```
new Student(“Lisa Studywoman”, “u1234567”, “BAC”)
```

To use, access fields:

```
Student s = …;
println(s.name()+” studies “+s.program());
```


The Design Recipe

Step 2. Function Signature and Purpose Statement

Australian National University

Recall

Function Signature

```
void main() {
    println(“Hello World”);
}
```

Recall: Multiple Audiences

Java uses the Java signature of your function to check your code.

The Java signature can only be on the level of granularity that Java's type system supports.

E.g.:

Are there negative sizes? Java thinks it's possible

```
int getSize(Item item) {}
```


Signature & Purpose Statement

Purpose Statement

```
/** pizzaShares returns how much of pizza (in surface area)
  * with a given diameter everyone in a group of a given size
  * gets when dividing the pizza evenly.
  * @param diameter – the diameter of the pizza in inches,
  * a positive number
  * @param groupSize – the number of people in the group > 0
  * @return how many square inches of pizza (rounded down) every
  * person in the group will get (>= 0)
  */
int pizzaShares(int diameter, int groupSize) {
}
```

Human Signature

Java Signature

The Design Recipe

Step 4. Design Strategy

(yes, we skipped Step 3 until next week)

Australian National University

Design Strategies

Your goal is to always define small, simple functions that do at most one thing. If things get too complicated, think of a helper function that can do part of the job, and add it to the wishlist.

Design Strategies help you keep some discipline around this.

Design Strategy: Simple Expression

For when the code will just be a simple expression

```
/**
 * Adds up two numbers
 * Examples: [skipped]
 * Strategy: Simple Expression
 * @param number1 – the first number
 * @param number2 – the second number
 * @return the sum of number1 and number2
 */
int add(int number1, int number2) {
  return number1 + number2;
}
```

New in Step 4
Describes the shape of Step 5

Happens in Step 5

Design Strategy: Combining Functions

For when the code will just be a combination of function calls

```
/**
 * Determines whether an integer is square
 * Examples: [skipped]
 * Strategy: Combining Functions
 * @param integer – the integer that is possibly square, >0
 * @return true if integer is square, false if not
 */
boolean isSquare(int integer) {
    int intRoot = RoundInt(Sqrt(integer));
    return Equals(integer, intRoot * intRoot);
}
```

New in Step 4
Describes the shape of Step 5

Happens in Step 5

Design Strategy: Case Distinction

For when a number of cases need to be distinguished without an underlying data definition.

```
/** … * Strategy: Case Distinction … */
TrafficLightState StringToTLS(String str) {
  return switch(str) {
    case “red” -> TrafficLightState.RED;
    case “yellow” -> TrafficLightState.YELLOW;
    case “green” -> TrafficLightState.GREEN;
    default -> DefaultCaseError();
  }
}
```

Exclude this case in your human signature. Java requires it to be there.

Design Strategy: Template Application

For when you need to apply a template.

```
/** … * Strategy: Template application for TrafficLightState…*/
TrafficLightState Rotate(TrafficLightState tls) { …
  return … switch(tls) {
    case RED -> …;
    case YELLOW -> …;
    case GREEN -> …;
  } …;
}
```

This is the state after step 4.
We copied the template from the data definition, and removed the //s .

Combining Design Strategies

- Only ever use ONE template application or case distinction in a single function (and not both).
- You can combine either case distinction or template application with simple expressions and/or combining functions (and the latter two with each other) within reason. Your code needs to stay readable!
- The priority ranking is Template Application ~ Case Distinction > Combining Functions > Simple Expression. When documenting your chosen strategy, use the highest-ranking one that applies.
- While records have a template, it is fine to see using that as combining functions.