UNIVERSITY *of* TASMANIA
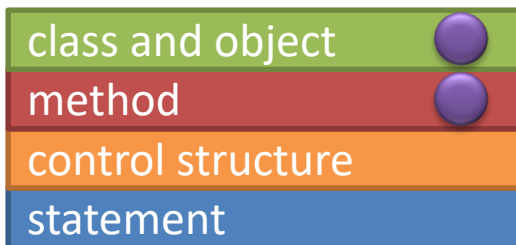
Object-oriented Program Design

Week 8

class and object
method
control structure
statement

Creating Your Own Classes

11 Creating Your Own Data Types
12 More Object Orientation
13 Functional Decomposition
14 Object-oriented Program Design

Dr James Montgomery, james.montgomery@utas.edu.au

8.1PP Programming Principles

- In about 2 pages of text, code snippets and images (if you want) explain some key programming and Java-related terms in your own words
- Do not be frightened. You need to write so we can understand you, not so you'll win prizes

8.2CR Debug This

- Given some code containing errors and clues left by its mysterious author, identify and fix the errors, documenting what you did, what clues you followed, and what you changed

8.3DN Object-Oriented Collection Manager

- Transform your 7.1PP program (and 6.2CR class definition) to be more object-oriented (which is the topic of today's lecture)
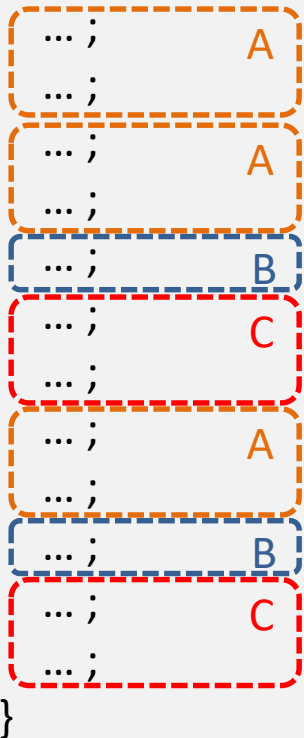
Monolithic application

```
main() {
    ... ;          A
    ... ;
    ... ;          A
    ... ;
    ... ;          B
    ... ;          C
    ... ;
    ... ;          A
    ... ;
    ... ;          B
    ... ;          C
    ... ;
}
```

Procedural programming

```
void a() { ... }

void b() { ... }

void c() { ... }

main() {
    a();
    a();
    b();
    c();
    a();
    b();
    c();
}
```

Object-oriented programming

```
main() {
    o.a();
    o.a();
    o.b();
    p.c();
    o.a();
    o.b();
    q.c();
}
```

Object o
(class AB)

void a() { ... }

void b() { ... }

Object p
(class C)
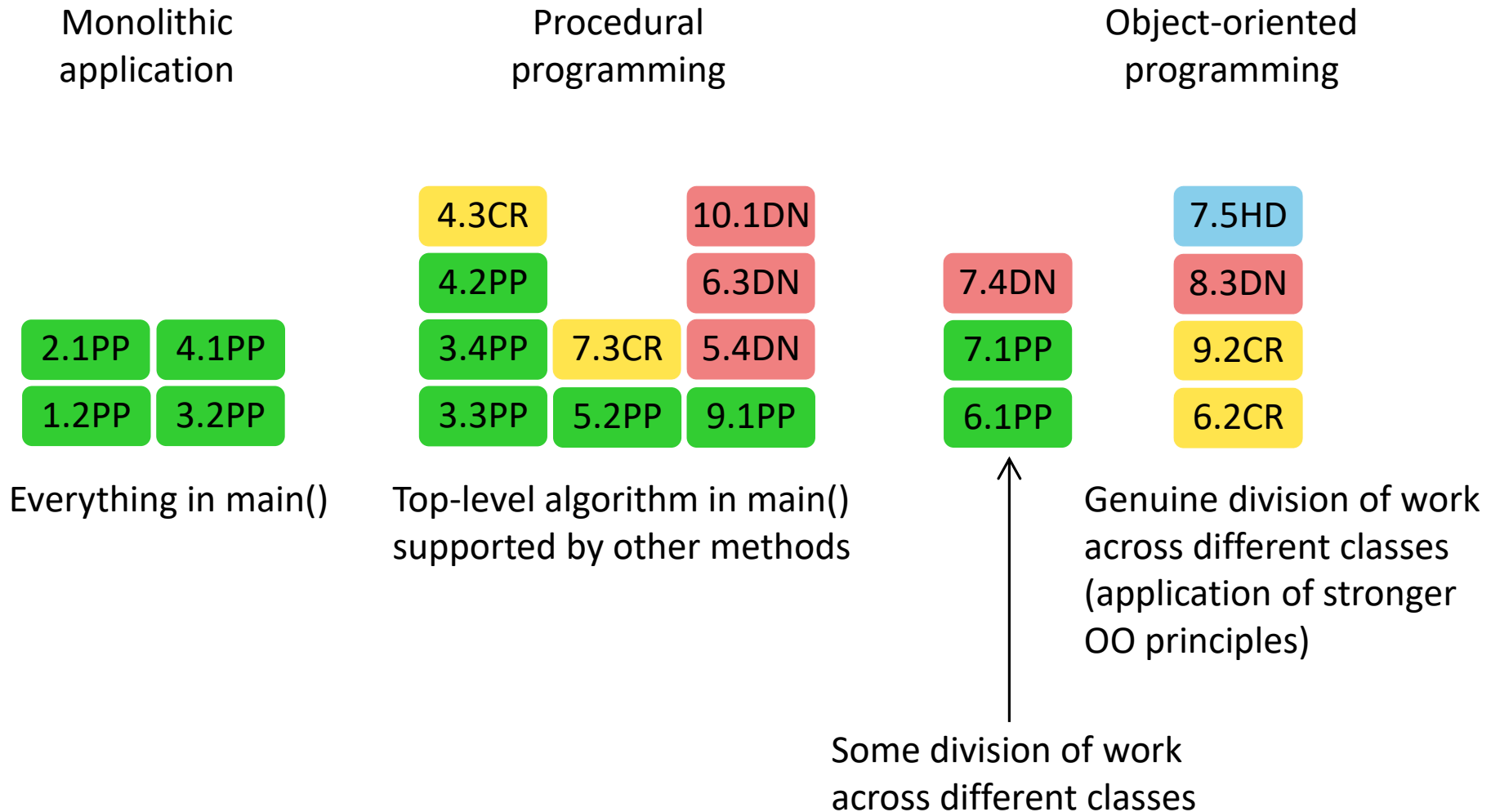
void c() { ... }

Object q
(class C)

void c() { ... }

*Increasing **separation of concern**. Different parts of the software take greater responsibility for solving smaller parts of the overall problem.*

Programming tasks in this unit

Monolithic application

Procedural programming

Object-oriented programming

4.3CR			10.1DN
4.2PP			6.3DN
3.4PP	7.3CR	5.4DN	
3.3PP	5.2PP	9.1PP	

2.1PP	4.1PP
1.2PP	3.2PP

7.5HD
8.3DN
7.4DN
9.2CR
7.1PP
6.2CR
6.1PP

Everything in main()

Top-level algorithm in main() supported by other methods

Genuine division of work across different classes (application of stronger OO principles)
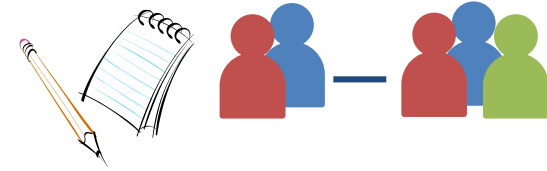
Some division of work across different classes

*Increasing **separation of concern**. Different parts of the software take greater responsibility for solving smaller parts of the overall problem.*
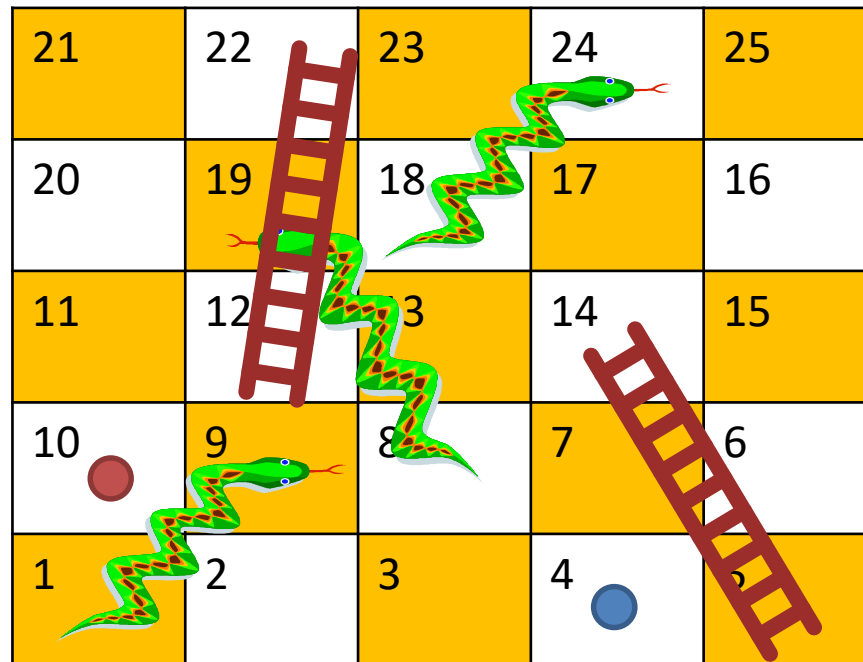
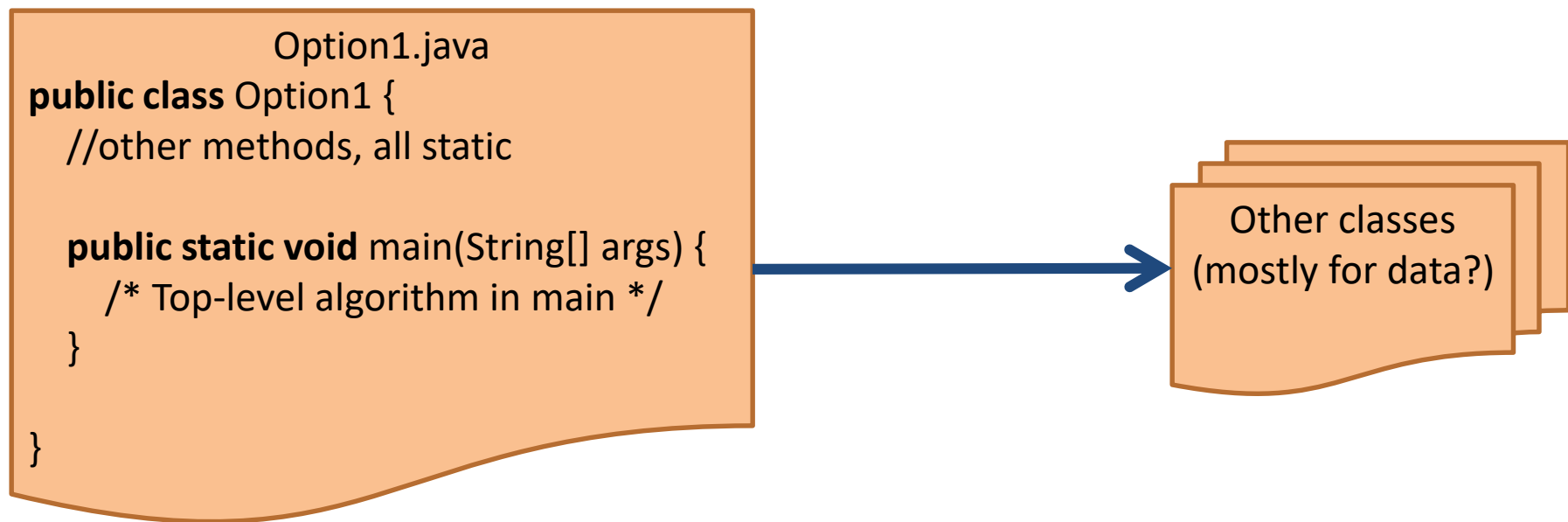*There is an accompanying self-guided activity to implement the game*

Task: What kinds of objects could we model in this game? What kind of attributes and abilities might they have?

Single source → interacting objects

Use other objects, but keep the top-level algorithm in main() and other static methods of that class

```
                     Option1.java
public class Option1 {
   //other methods, all static

   public static void main(String[] args) {
      /* Top-level algorithm in main */
   }

}
```

Other classes (mostly for data?)

Keeps top-level algorithm in one place, but can become complex to manage, especially as the complexity of the data model grows

Overall is less object-oriented

Single source → interacting objects

Using an 'organiser' class

Option2.java
```
public class Option2 {
    public static void main(String[] args) {
        Organiser o = new Organiser();
        o.topLevelAlg();
    }
}
```

Organiser.java
```
public class Organiser {
    //instance data shared by top-level alg

    //other methods

    public void topLevelAlg() {
        /* Top-level algorithm here */
    }
}
```
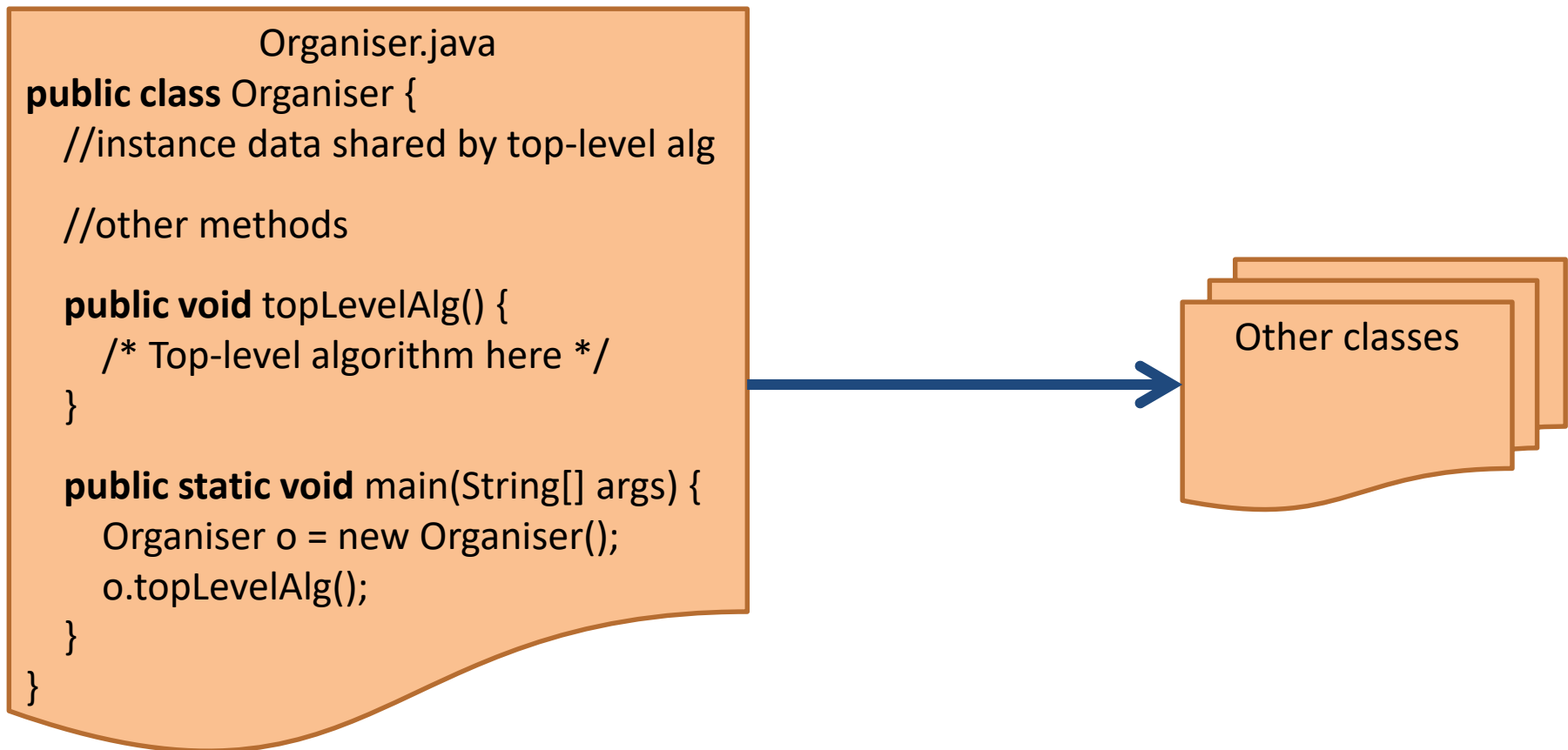
Other classes

- Organiser's code can be reused in different settings (different main programs)
- Data shared by many methods can be made 'global' (within Organiser)
- Facilitates change from console to graphical application
- But one more file and may seem like overkill for simple programs

*See Task 8.3DN Object-oriented Collection Manager*

Using an 'organiser' class with in-built main()

Organiser.java

```java
public class Organiser {
    //instance data shared by top-level alg

    //other methods

    public void topLevelAlg() {
        /* Top-level algorithm here */
    }

    public static void main(String[] args) {
        Organiser o = new Organiser();
        o.topLevelAlg();
    }
}
```

Other classes

Option 3 is if you really hate additional files; done if you've decided exactly how your program is going to be used, since a little less flexible

'Library' versus 'Organiser' classes
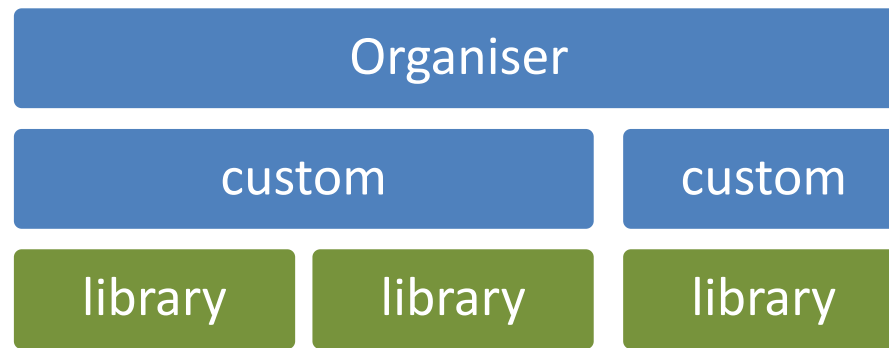
Library class	Organiser class
• the low-level 'components' of your software • general purpose (or common enough to allow reuse)	• defines the top-level algorithm • problem-specific • may use other problem-specific (custom) classes and library classes

(This is a *useful* distinction; these are not labels you will need to memorise)

Suitable modules for 'library' classes

Organiser

custom | custom

library | library | library

Constructor

One method for **each 'action'** the object can perform

• plus any necessary methods to do the separate parts of the action

Setters for instance variables

• if needed/appropriate

Getters for instance variables

• if needed

A **trace()** method

```
private void trace(String message) {
    System.out.println(message);
}
```

Not *required* by Java, but useful to have

• for use during development

Suitable modules for 'organiser' classes

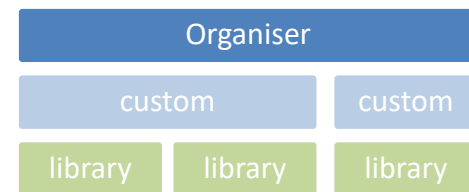| Organiser		
custom		custom
library	library	library

Constructor

- if necessary to do any set up

One method for each sub-problem

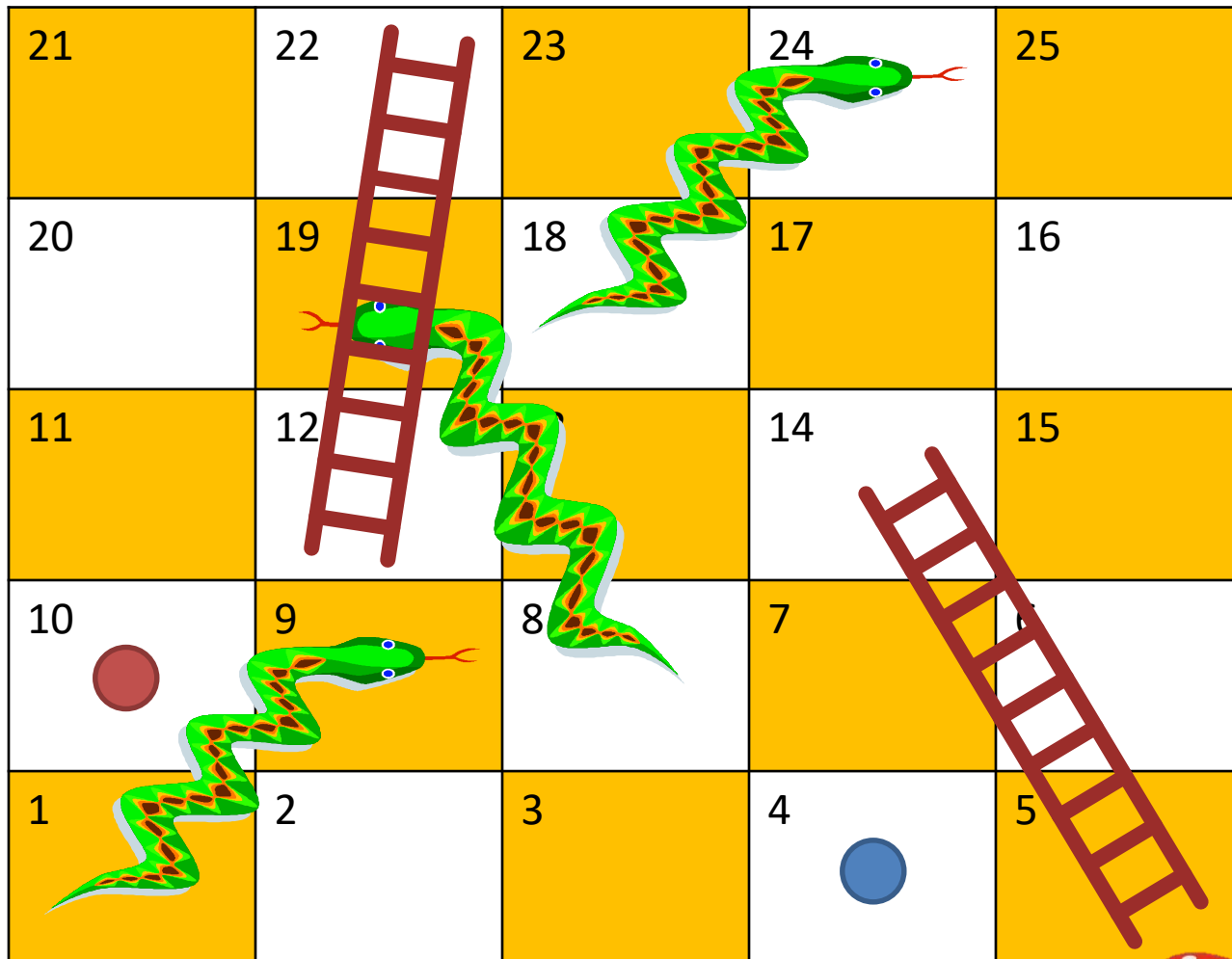- Often these relate to:
  - Input
  - Processing
  - Output

Any methods needed for separate parts of each sub-problem

A trace() method

- for use during development
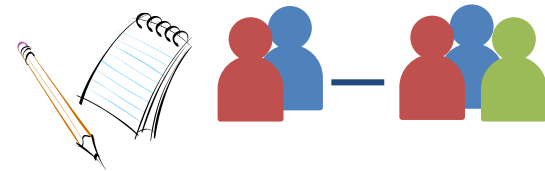
Partial top-level algorithm

- create any other components (objects)
- ask if user wants to play
- while user wants to play
  - play one game
    - set player positions to 0
    - set current player
    - while the game is not over
      - current player moves
      - if game not yet over
        - swap players
    - announce winner
  - see if user wants to play again

Task: Assume we've drafted the top-level algorithm as above

Sketch a structure chart of a possible breakdown into different tasks

Note: You can largely ignore parameters and return types; think in terms of responsibilities

Top-level algorithm

Create **Die** and **Board** objects
SALGame()
constructor

Ask if user wants to play
startPlaying()

While user wants to play

    Play one game
playGame()

        Set player positions to 0

        Set current player

        While the game is not over

            Current player moves
makeAMove()

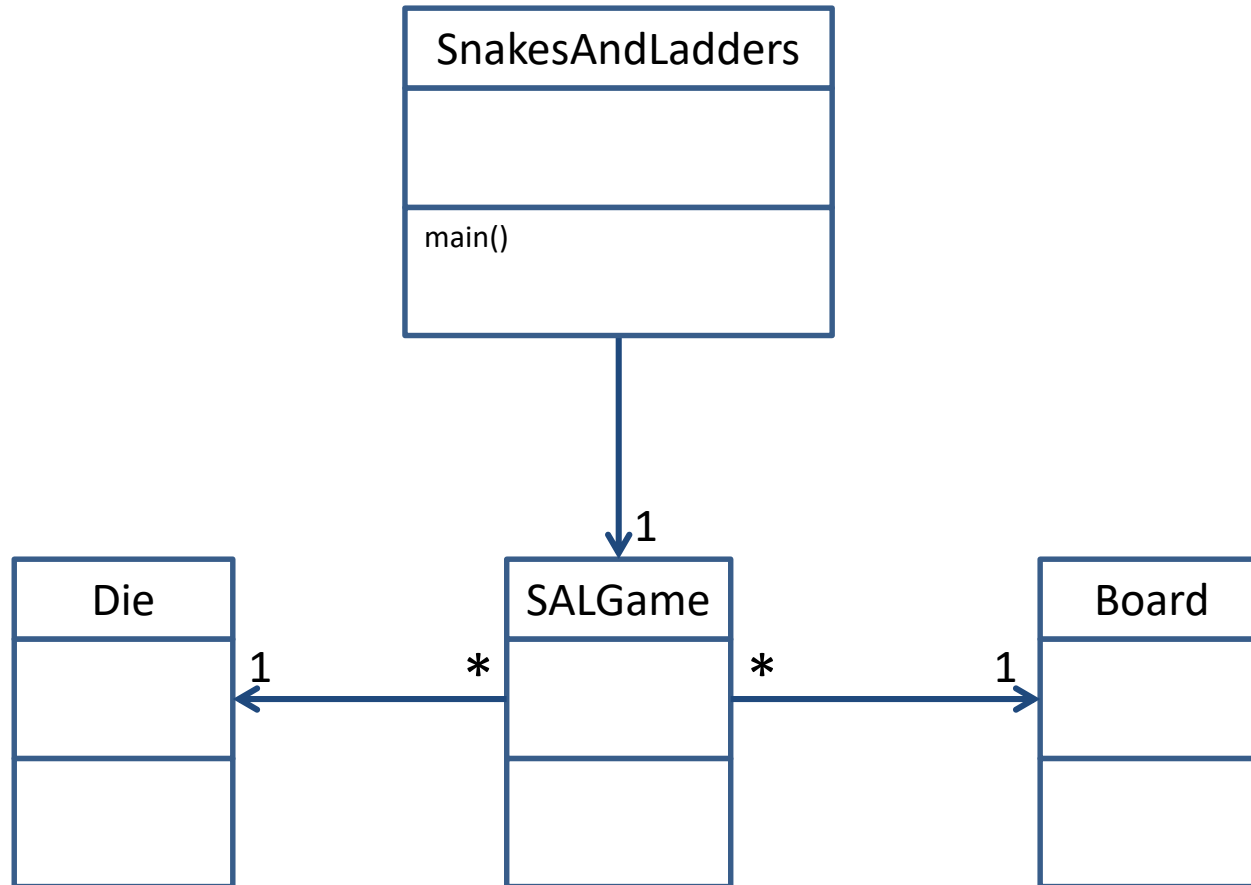   If game not yet over

                Swap players
swapPlayer()

        Announce winner
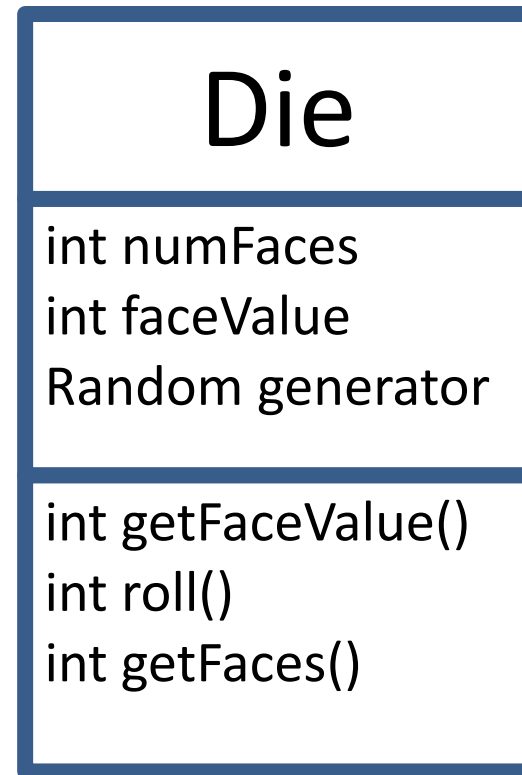
See if user wants to play again

Snakes & Ladders classes

*One, relatively
simple approach*

```
          ┌─────────────────────┐
          │  SnakesAndLadders   │
          ├─────────────────────┤
          │                     │
          ├─────────────────────┤
          │  main()             │
          │                     │
          └─────────────────────┘
```

| Die | SALGame | Board |

SnakesAndLadders — 1 → SALGame

Die 1 ← * SALGame * → 1 Board
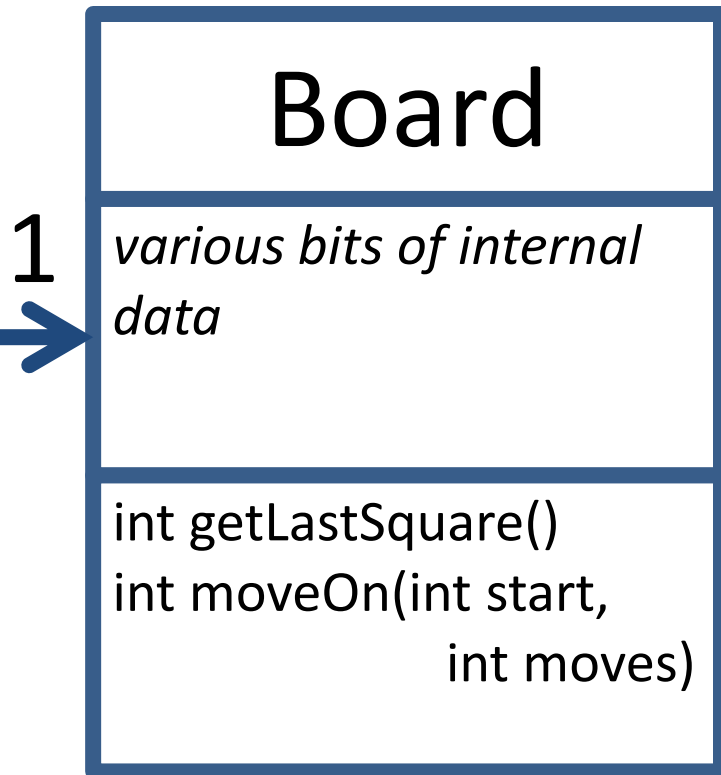
If you've ever studied databases then this should look familiar

Die is responsible for creating a virtual six-sided game die, holding its state and randomising the upward facing side

Die

int numFaces
int faceValue
Random generator

int getFaceValue()
int roll()
int getFaces()

1

Board

1

various bits of internal data

int getLastSquare()
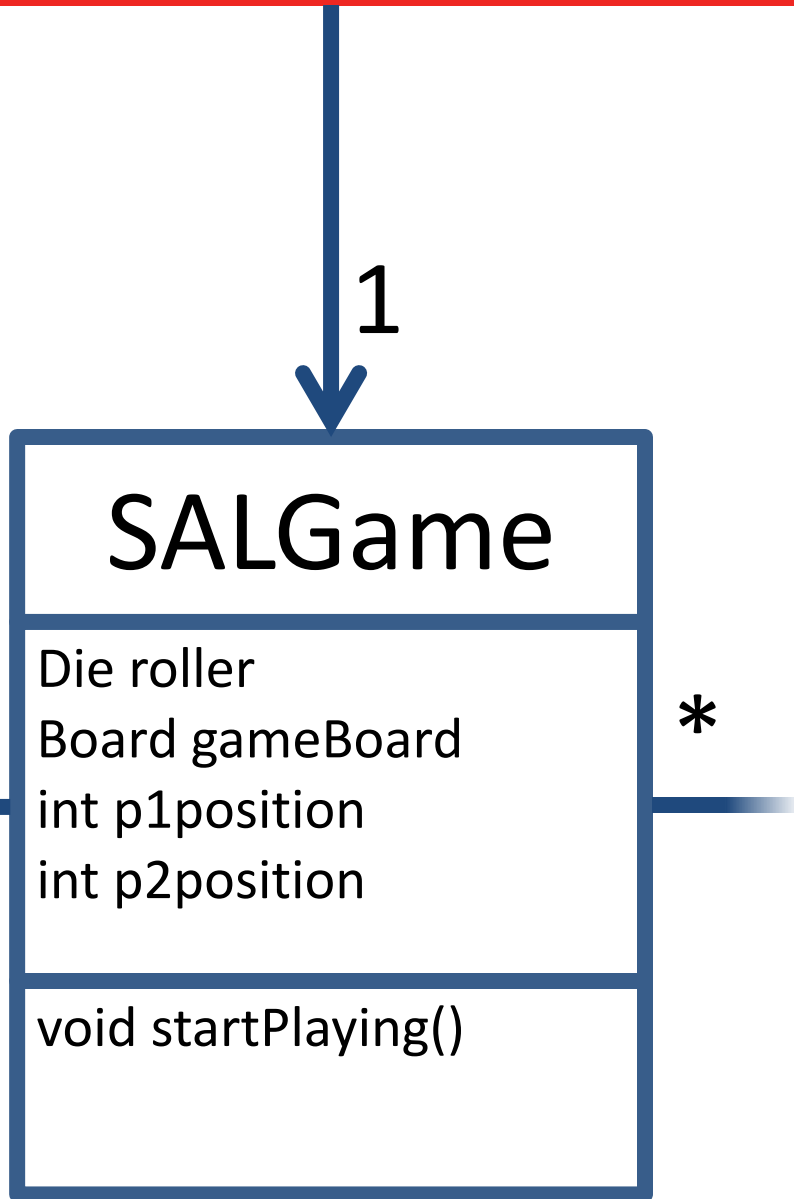int moveOn(int start,
                    int moves)

Board knows how many squares there are, where the snakes and ladders begin and end and can tell you where you'll end up after a given number of moves starting from a particular square

SALGame

Die roller
Board gameBoard
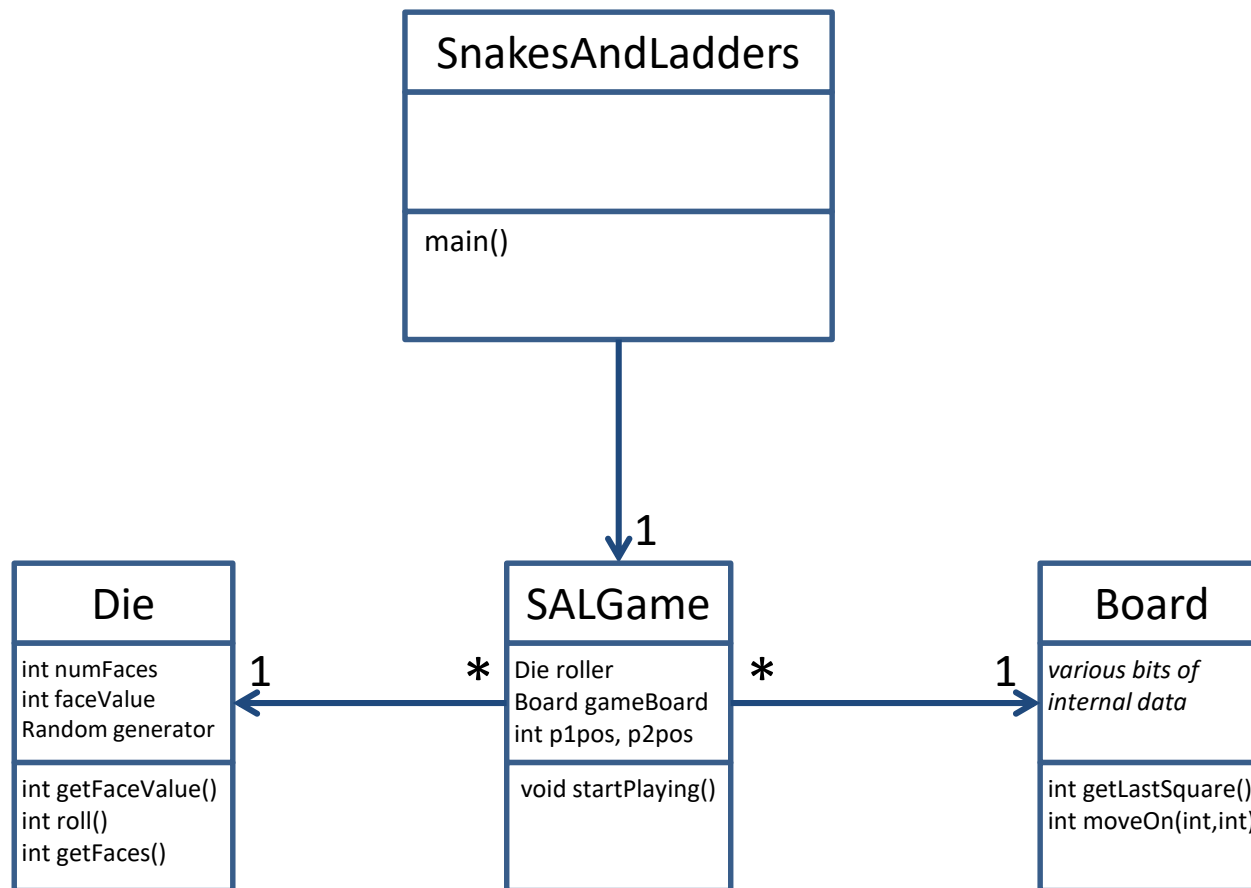int p1position
int p2position

void startPlaying()

1

*

*

Top-level algorithm
- create **Die** object
- create **Board** object
- declare **int** variables for player positions
- see if user wants to play
- while user wants to play
  - play one game
  - see if user wants to play again

Later: download and work through the self-guided development of the Snakes and Ladders game yourself

Let's implement it

Top-level algorithm

Create **Die** and **Board** objects — SALGame() constructor

Ask if user wants to play — startPlaying()

While user wants to play

Play one game — playGame()

Set player positions to 0

Set current player

While the game is not over

Current player moves — makeAMove()

If game not yet over

Swap players — swapPlayer()

Announce winner

See if user wants to play again