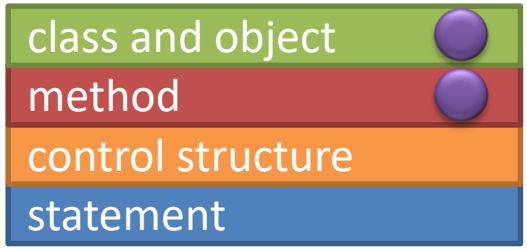
Object-oriented Problem Solving & Functional Decomposition

Week 7



Creating Your Own Classes

11 Creating Your Own Data Types
12 More Object Orientation
13 Functional Decomposition
Appendix: Documentation

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

Tasks starting this week

7.1PP Arrays of Objects

- Create a record management program using your 6.1PP class
- **This is the most substantial pass-level program you will write**

7.2PP Structure Charts

- Create a structure chart documenting your 7.1PP program

7.3CR Custom Program Design

- Create the initial design for a program of your own, demonstrating use of a custom data type and arrays. Get your friends' and tutor's feedback on your idea

7.4DN Custom Program

- Implement it! (So not really *this* week)

7.5HD Improved Custom Program

- Get our advice on how your program can be improved: add some more complexity and improve the internal structure and use of abstraction

Software is complex

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.*

How complex is the Pass level?

```
public class Program {

    … m1(…) { … }

    … m2(…) { … }

    … main(…) {
        m1(…);
        m2(…);
        m1(…);
    }
}
```

```
public class DataA {
    private type property;
    //constructor
    //getters and setters
    public String toString() { … }
}
```

```
public class DataB {
    private type property;
    //constructor
    //getters and setters
    public String toString() { … }
}
```

Able to apply **functional decomposition** to break down software components into multiple methods in source file containing main()

Able to **use arrays**

Able to **plan and document**

Able to **define new data types**

Credit level applies same skills with **greater sophistication**

Distinction and above?

```
public class Program {

   … m1(…) { … }

   … m2(…) { … }

   … main(…) {
      Component c;
      c = new Component();
      m1(…);
      m2(…);
      m1(…);
      c.doSomething();
   }
}
```

```
public class DataA {
   private type property;
   //constructor
   //getters and setters
   public String toString() { … }
}
```

```
public class Component {
   private type internalData;
   //constructor
   //(maybe) getters & setters
   //other methods to help
   //  solve part of the problem
}
```

Starting to use class mechanism to split *functionality* across different source files

HD level applies same skills to **larger problems** & with **greater sophistication**

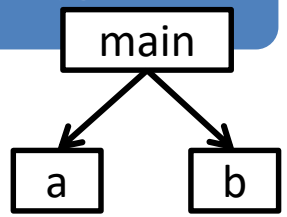
Today's topics

Object oriented problem solving *in general*

Defining a new self-sufficient data type

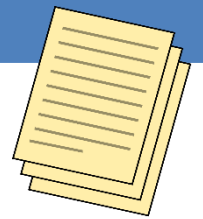
- Implementing a game die

Functional decomposition and structure charts



- making problems manageable

Documentation



- why we make you do it

Object-oriented problem solving

Understand the problem


Die


Board

Dissect the problem into manageable pieces

Determine the **objects** that need to be modelled

Ask: can these be modelled with an existing class or primitive type? (saves effort, reduces complexity)

Object-oriented problem solving

Understand the problem


Die


Board

Dissect the problem into manageable pieces

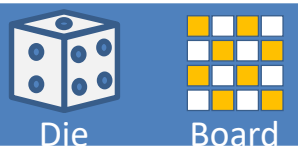
Design a solution

What **properties** will the objects need?

How will the objects **interact**? (What methods will they each have and call on each other?)

Object-oriented problem solving

Understand the problem



Dissect the problem into manageable pieces

Design a solution

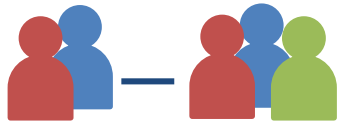
Consider alternatives to the solution and refine it

Implement the solution

Write and test new classes

Create objects: write and test solution to problem

Modelling a game die



Task: What *attributes* and *behaviour* do we need to model a die?

Constraints: Want to be able to create dice with different numbers of sides

A Die

Attributes (data)

- Number of sides
  - default?
  - determined by client?
- Current face value
  - Default starting value?
  - Starting value determined at random?
- *n*=? Random number generator (for roll)
- ‘Knowledge’: minimum possible faces

Behaviour (methods)

- Construct Die (initialisation)
  - How many sides?
  - Minimum possible number? (handle this)
- Getters?
  - Get current face
- Setters?
  - none needed
- Doers?
  - Roll die
    - changes current face
    - could also return new face

Diagrammatic representation of a Die

The class

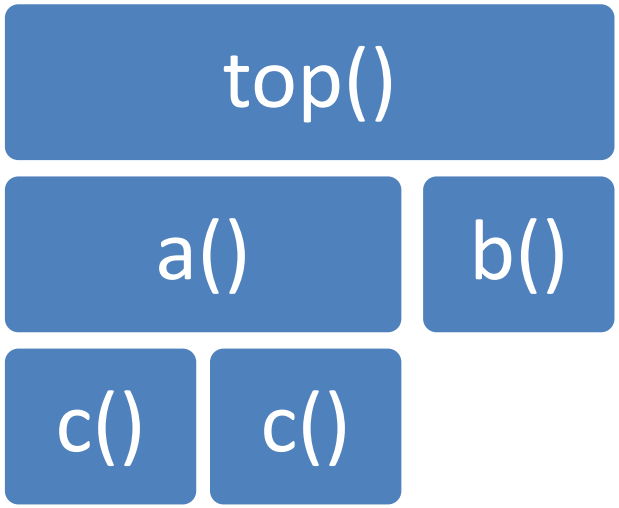
Die
int numFaces int faceValue Random generator
int getFaceValue() int roll() int getFaces()

Example object

dotty: Die
numFaces = 6 faceValue = 1 generator
int getFaceValue() int roll() int getFaces()

See Die.java and DieTester.java

Functional Decomposition
(Method)

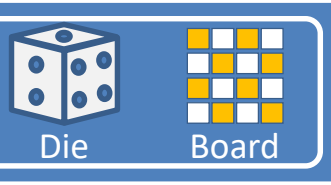



```
public class A {
    public void top() {
        a();
        b();
    }
    private void a() {
        c();
        c();
    }
    private void b() {
        …
    }
    …
}
```

Organise code into modules (methods)

'module' ≡ 'function', 'procedure', or *method*

Problem solving steps 2 & 3:



Dissect the problem into manageable pieces

Design a solution

Divide the problem into sub-problems and write a method to solve each sub-problem

Guiding principle: each method does *one* thing only

Advantages

- ✓ Related code found together
- ✓ Can solve the problem a bit at a time
- ✓ Can reuse parts of solutions to solve other problems

Examples from the tasks

4.2 Typing Tutor	5.2 Collection of Strings	6.2 Objects w More Abilities
main() calls **printHeading** **runTutorial**	**main**() calls **add** **printList** **averageLength**	**main**() calls **read*TypeName*** which calls **read*EnumName*** which may call **promptForInt**

each of which helps solve *one* part of the problem

Sharing data between methods

Within an object	Between objects *or* Between main & methods	Within an object (but more flexibly)
Methods have direct access to all instance data, and so… …can read it …can modify it	Method parameters for data *in* Method return values pass data *out*	Parameters allow method reuse with different data

```
public class A {
    private int i;

    public int add(int n) {
        i += n;
        return i;
    }
    …
}
```

```
public class B {
    private A a = new A();

    public void next() {
        int x = a.add(1);
        …
    }
}
```

```
public class T {
    public void square(int s) {
        poly(4, s);
    }

    public void poly(int n, int s) {
        …
    }
}
```

ProTip: Method stubs

Add all planned methods as methods stubs first

- Everything for the method *except* the code inside its { } block

Put a single comment inside stub to say what's left to do

```
/** Moves the Turtle the given distance; will draw a line if the pen is down. */
public void move(int dist) {
   //TODO Implement this method
}
```

If method returns a value, add a statement to return *something*

```
/** Returns a string containing only letters a-z from message, all in upper case. */
public String cleanText(String message) {
   //TODO Remove non-letters, convert to upper case, return the result
   return ""; //dummy return value until method is actually implemented
}
```

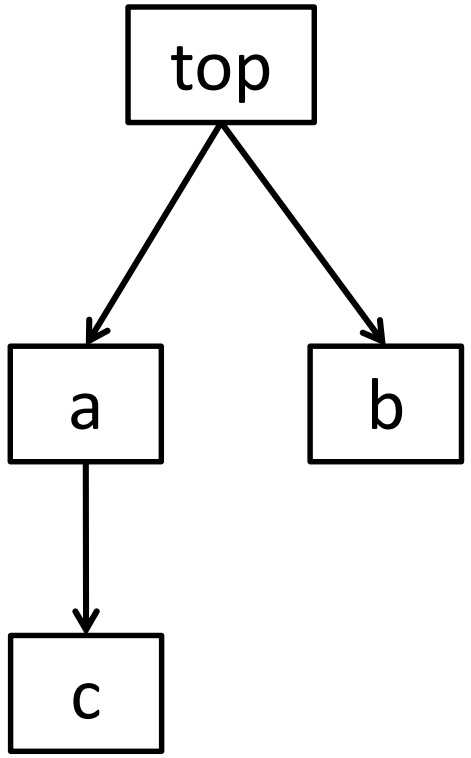
Structure Charts

A graphical depiction of the breakdown of a solution into smaller tasks (so actually general purpose)

You will use to show which methods call which other methods

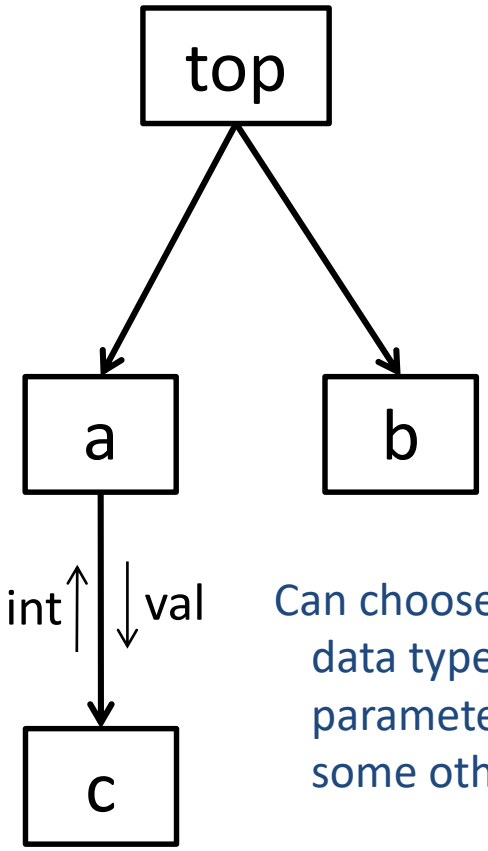
Can be used as a planning tool *or* to document an existing solution

```
public class A {
    public void top() {
        a();
        b();
    }

    private void a() {
        c();
        c();
    }

    private void b() {
        …
    }

    private void c() {
        …
    }
}
```



Show flow of data into and out of methods

```
public class A {
    public void top() {
        a();
        b();
    }

    private void a() {
        int num = 2;
        num += c(num);
        num += c(num);
    }

    private void b() {
        …
    }

    private int c(int val) {
        return val * 2;
    }
}
```

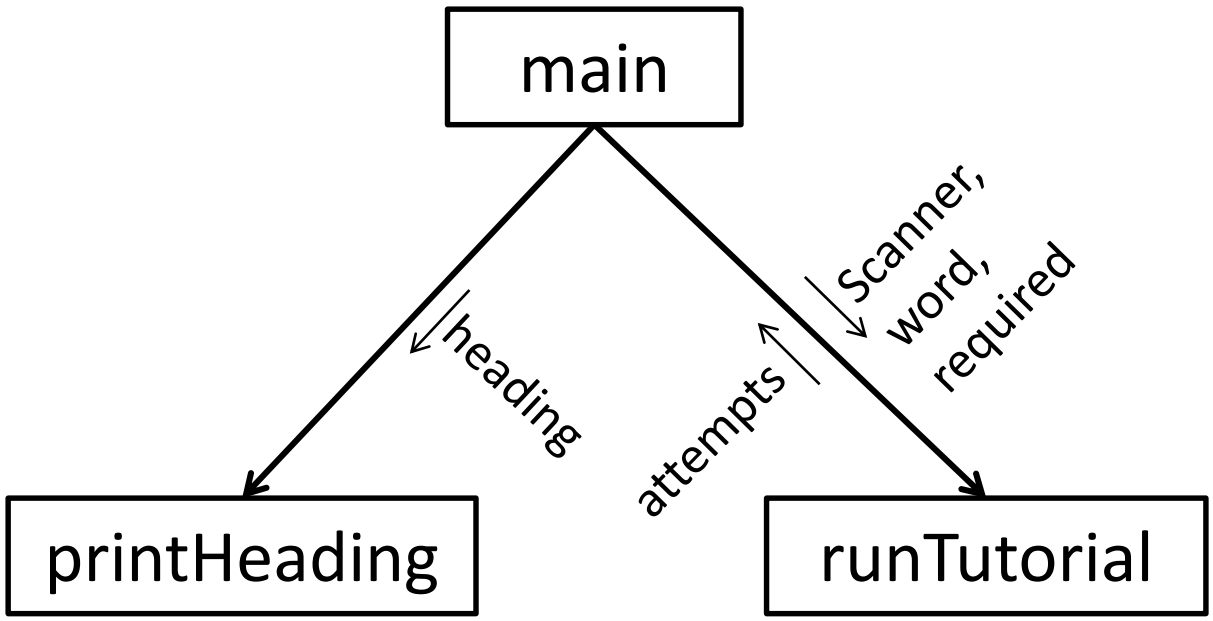


Can choose to label with:
- data type,
- parameter name, or
- some other meaningful name

From the tasks: 4.2 Typing Tutor

4.2
Typing Tutor

main() calls
printHeading
runTutorial



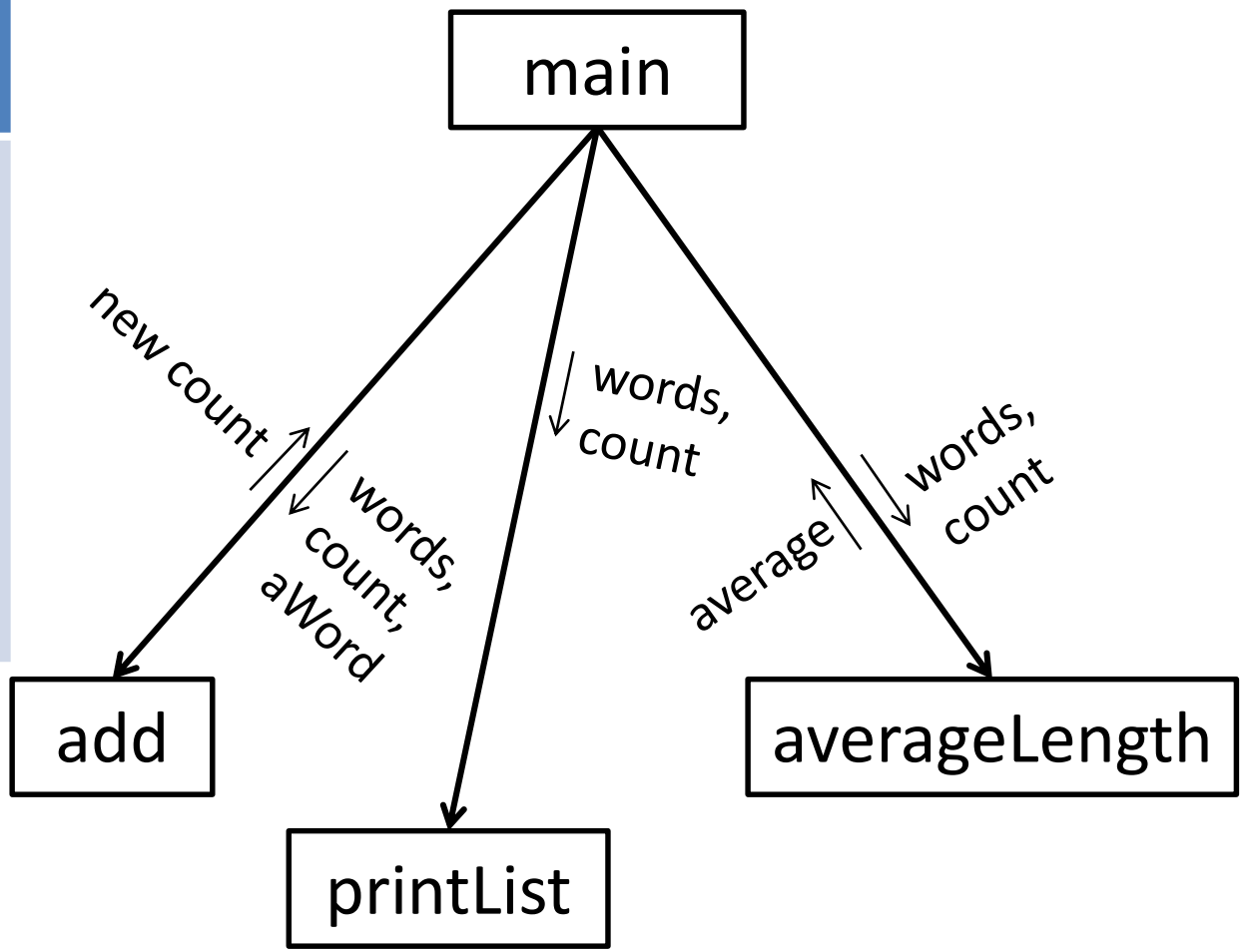
Tip: For data flow, can use a mixture of *type* names and *role* (parameter) names, whichever conveys the ideas more clearly

Consider the data going into runTutorial: *any* Scanner would do, but the String and int represent the word and required number of correct entries

From the tasks: 5.2 Collection of Strings

5.2
Collection of Strings

main() calls
add
printList
averageLength



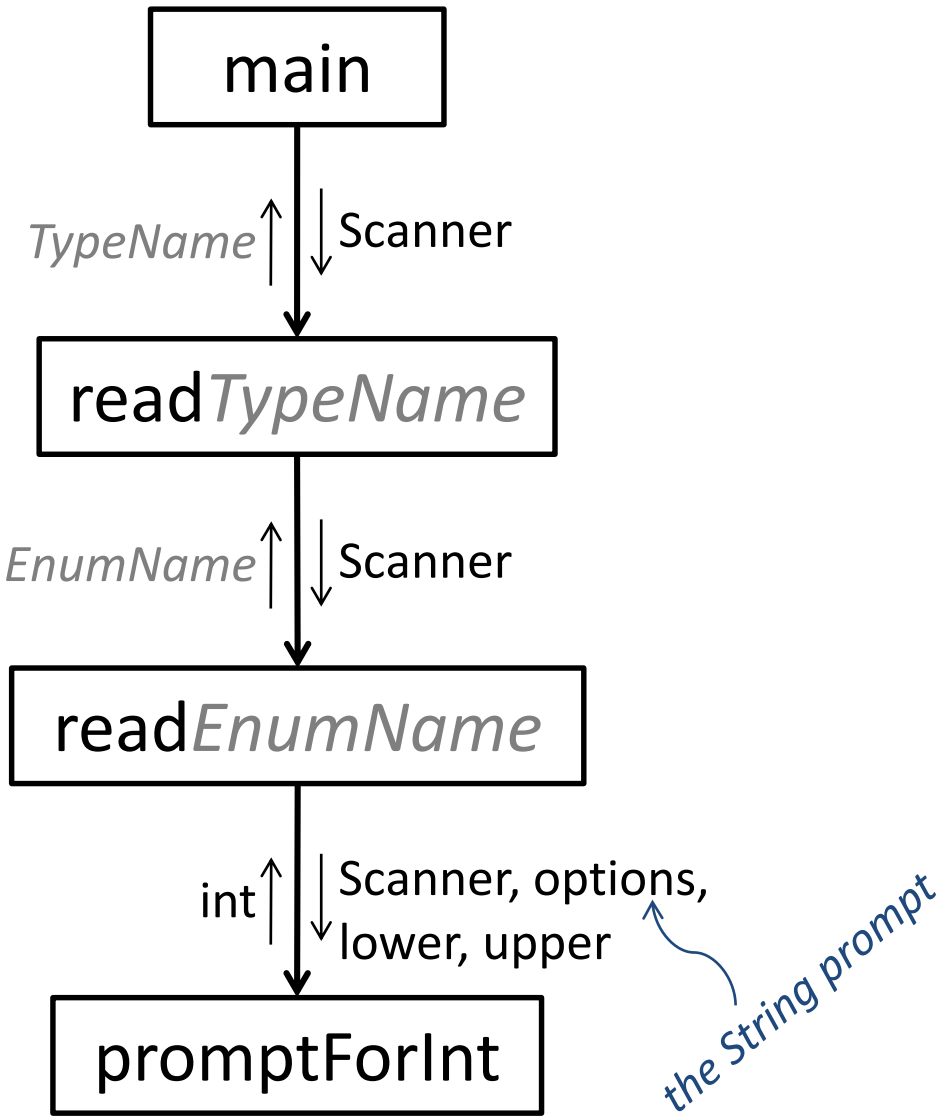
This structure chart uses a mixture of parameter names and short but meaningful labels for returned data

From the tasks: 6.2 Objects with More…

6.2 Objects w More Abilities

main() may call
read*TypeName*
which calls
read*EnumName*
which may call
promptForInt

Here we're mostly using type names because the read*Thing* methods are generating instances of a particular type, but we've also taken the liberty of relabelling the *prompt* parameter to make its role clearer



Demonstration: A Game of Chance

Highly abbreviated source code (download full version from MyLO)

```
boolean continuePlay(Scanner in, String prompt) { … }
```

```
void playGames(Scanner in) {
    do {
        playOneGame(in);
    } while (continuePlay(in, "Play again?"));
}
```

```
int playOneGame(Scanner in) {
    Die die = new Die();
    do {
            total += die.roll();
    } while (continuePlay(in, "Roll again?"));
    return total;
}
```

```
static void main(String[] args) {
    if (continuePlay(sc, "Roll the die?")) {
        playGames(sc);
    }
}
```

Player plays a number of rounds

In each round they roll a game die, adding to their **total**

If their **total exceeds 20** then they go bust and **lose (total – 20) points**

If their **total <= 20** after a roll they **may save** their winnings (**total / 4**)

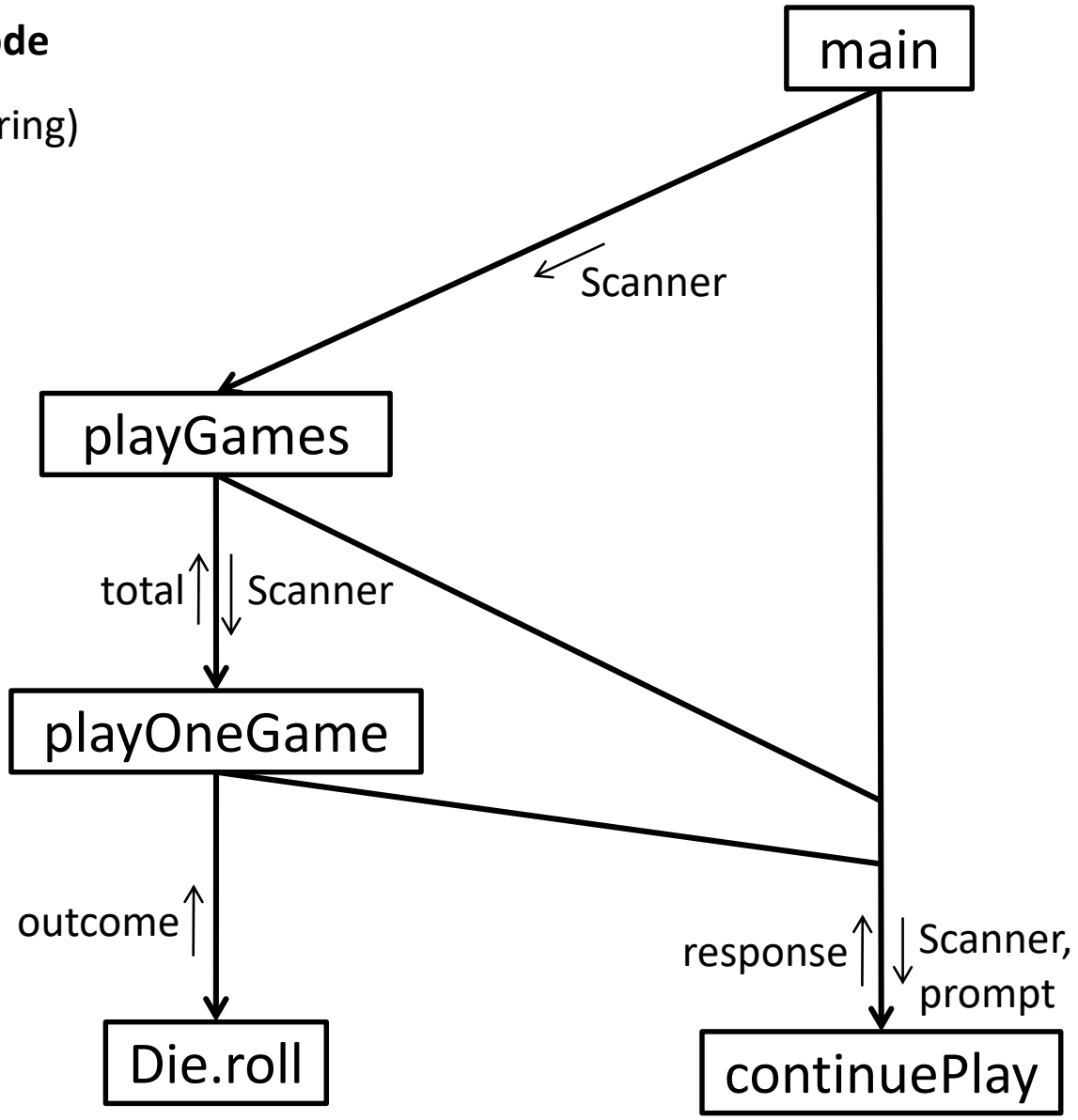
Each round's **total** is added to a **cumulative score**

See GameOfChance.java

Demonstration: A Game of Chance

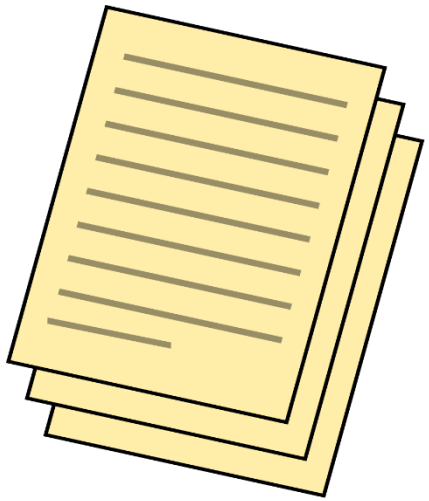
Even more abbreviated source code

```
boolean continuePlay(Scanner, String)

void playGames(Scanner) {
    playOneGame(in);
    continuePlay(in, "");
}

int playOneGame(Scanner) {
    die.roll();
    continuePlay(in, ""));
    return total;
}

static void main(…) {
    continuePlay(sc, "")

    playGames(sc);
}
```



Documentation

Why, what, how



Documentation

What is it?

- Record keeping
- “Instructions” on how to use your code

Why?

- Obligation
- Improvement

What to document

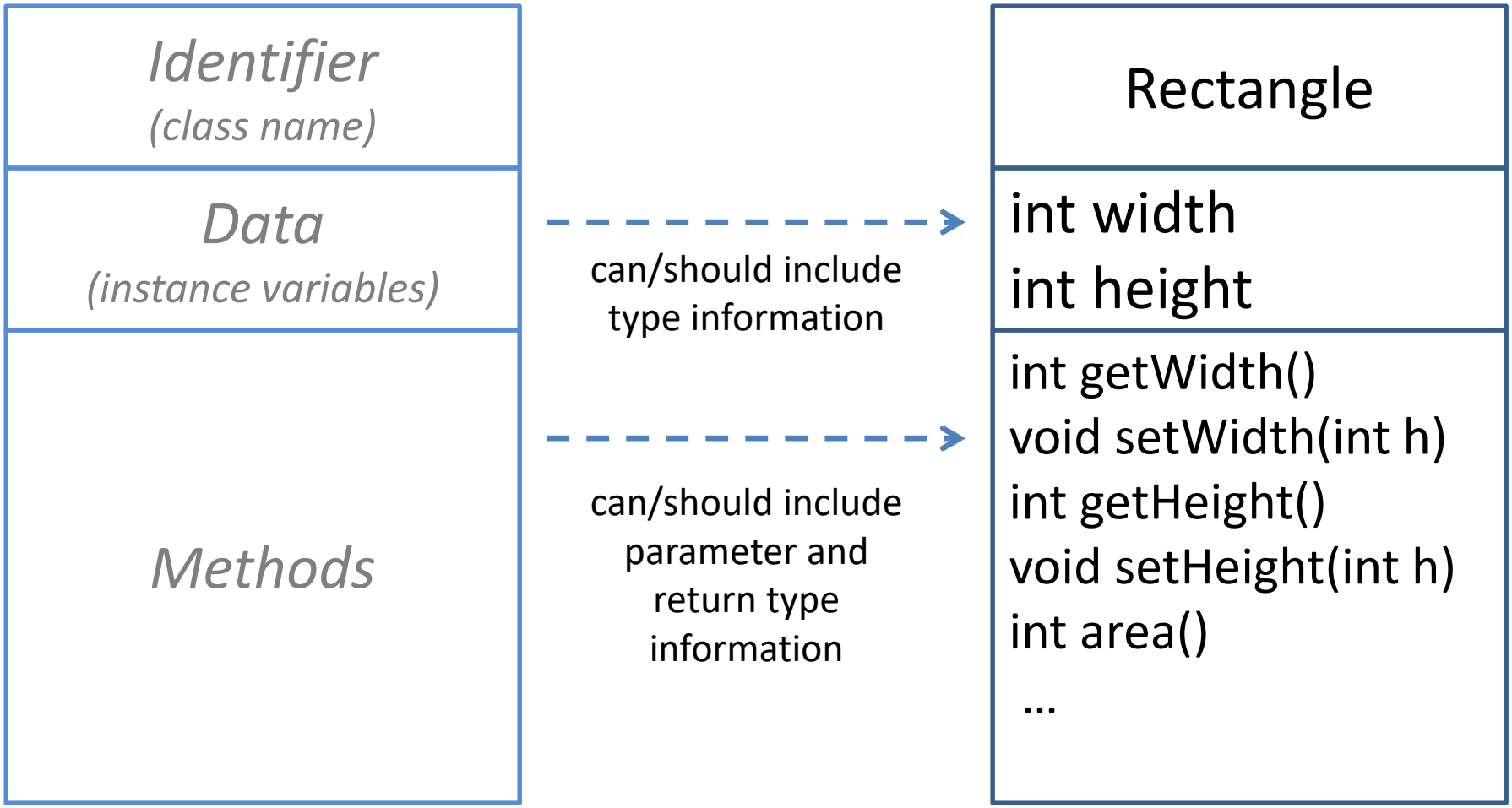
- Plan
- Program
- Process (software engineering)

How

- External
- Internal (comments)

Documenting a class

The 2-minute guide to UML and the **class diagram**



Documenting the main algorithm

In this unit, use informal ‘pseudocode’

Pseudocode

Variables:
int area1, *area of first paddock*
int area2, *area of second paddock*
Rectangle rect, *reused for each paddock*

Steps:
Create rect with width 20, height 40
Assign area1 rect’s area()
Change width and height of rect to 30, 10
Assign area2 rect’s area()
Print difference in area (area2 – area1)

Actual code

```
int area1; //area of 1st paddock
int area2; //area of 2nd paddock
Rectangle rect; //used for both

rect = new Rectangle(20, 40);
area1 = rect.area();
rect.setWidth(30);
rect.setHeight(10);
area2 = rect.area();
System.out.println(area1 - area2);
```

Documenting the code

Implicit

- Meaningful identifiers
- Comprehensible algorithms

Explicit (through comments)

- Strategic: what we are doing
- Tactical: how we are doing it (less common)

Commenting, what and where

```
/**
 * Description of the class: what it is for, what it does.
 * @author Author Name
 * @version Modification date(s)
 */
public class ClassName {
   private int someValue; //description of variable; what does it represent?

  /**
   * One-sentence description of what method does.
   * Any assumptions/requirements for it to work correctly.
   * @param x  description of x parameter
   * @param y  description of y parameter
   * @return description of what is returned
   */
   public int someMethod(int x, int y) {
      // description of what next block of statements does/achieves
      ...
      return x * y + 1;
   }
}
```

/** Is a multi-line *documentation* comment. */

Dark Italics represent places where you would write something different (more meaningful)