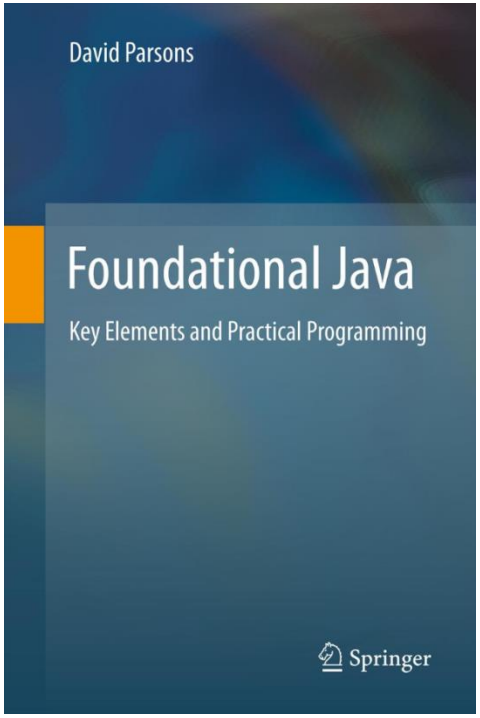


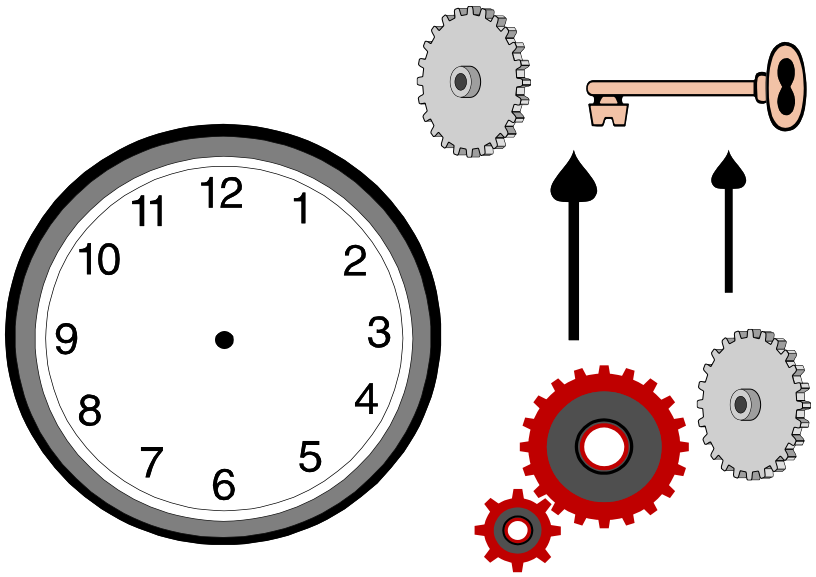
Chapter 7
Objects Working Together: Association, Aggregation and Composition

Foundational Java

Key Elements and Practical Programming

Components

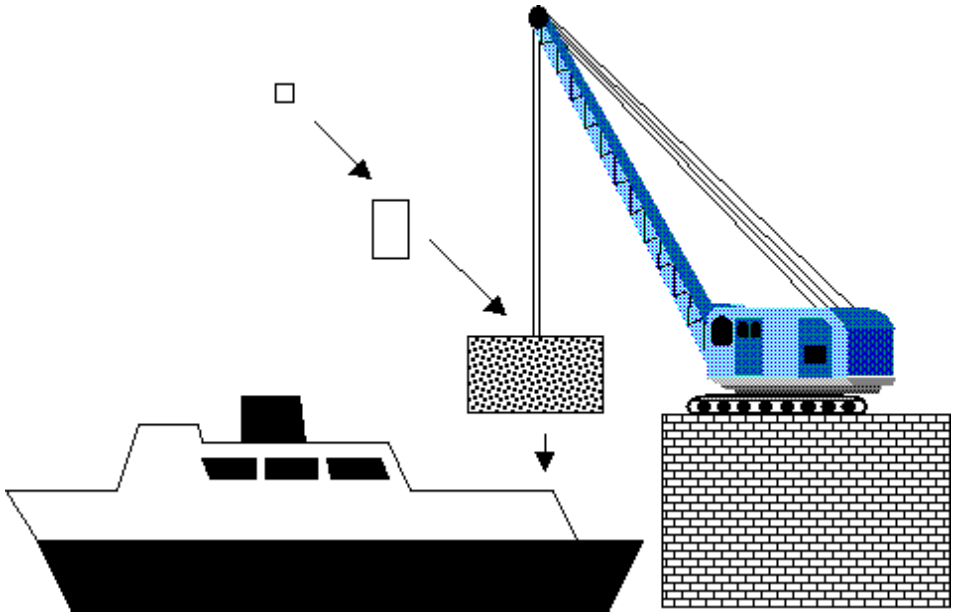
- Objects can be combinations of components



- Composition
  - Lifetime dependency
- Aggregation
  - Looser relationship

Aggregations as Collections

- Aggregations may combine objects in many layers

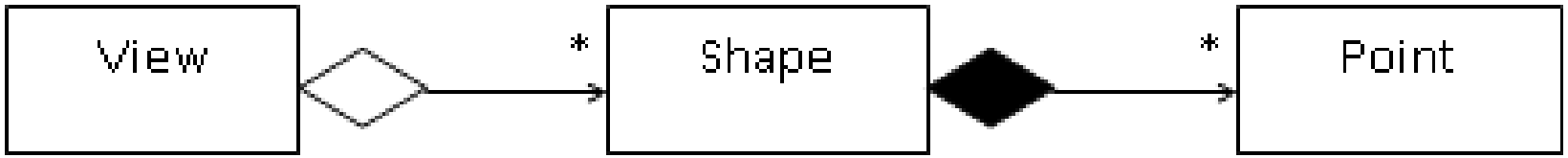


UML Notation

- Aggregation
  - Open diamond
- Composition
  - Filled diamond

* = zero to many

⟶ = unidirectional association



Message Passing

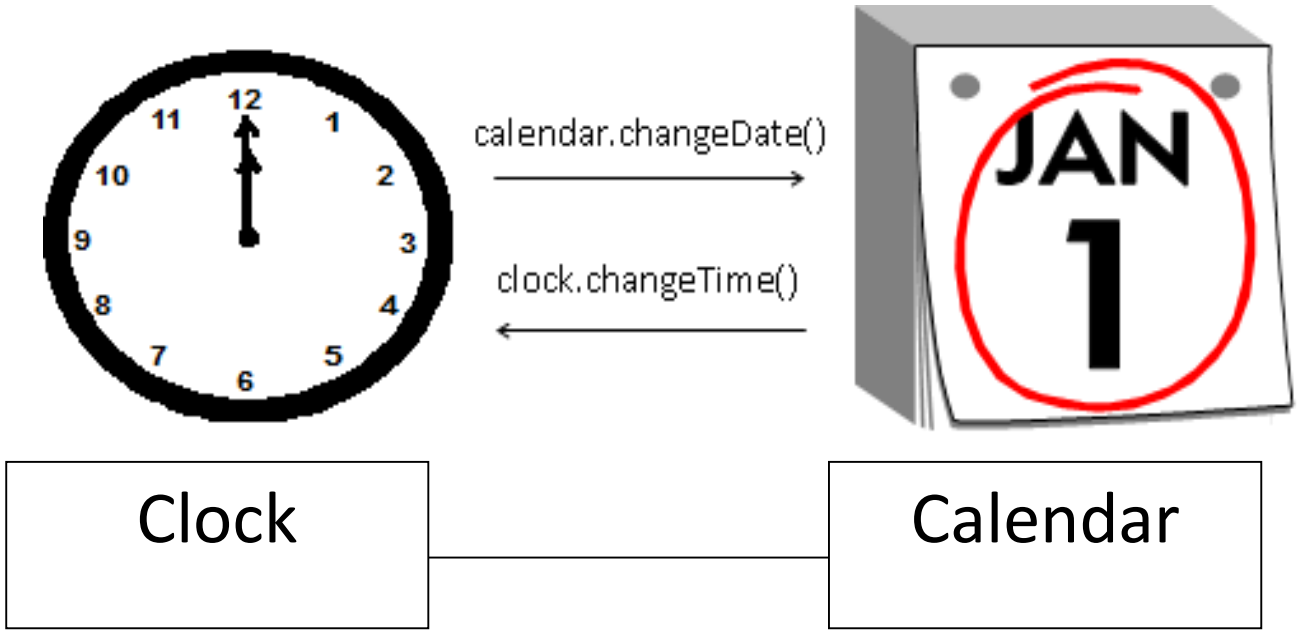
- In the real world, people and objects constantly interact
- Parts of the objects themselves also interact
- In object-oriented design:
  - We call people who interact with objects ‘actors’
  - We call the relationships between interacting objects ‘associations’
  - We talk about objects ‘passing messages’ to each other

Object Association

- Aggregation and composition are specific types of association
- An association is simply a mechanism for objects to communicate with each other
- Associations may be
  - one-to-one
  - one to many
  - many to many
- An object-oriented program consists of many objects collaborating to produce the required system behaviour

Association Direction

- Messages sometimes need to pass in both directions
  - Bidirectional association



Implementing Associations

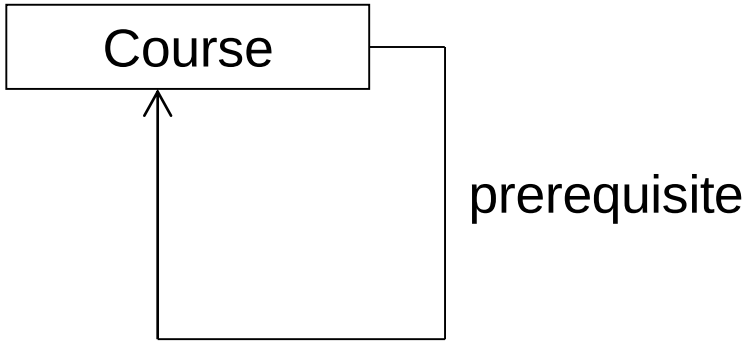
- Object references as fields


```
public class Course
{
  private CourseDelivery[] deliveries;
// can be created in the constructor
```

```
public class CourseDelivery
{
  private Location location;
// can be created in the constructor
```

Association to the Same Class

- Occasionally we need an association between objects of the same class
  - A Course may have another Course as a prerequisite



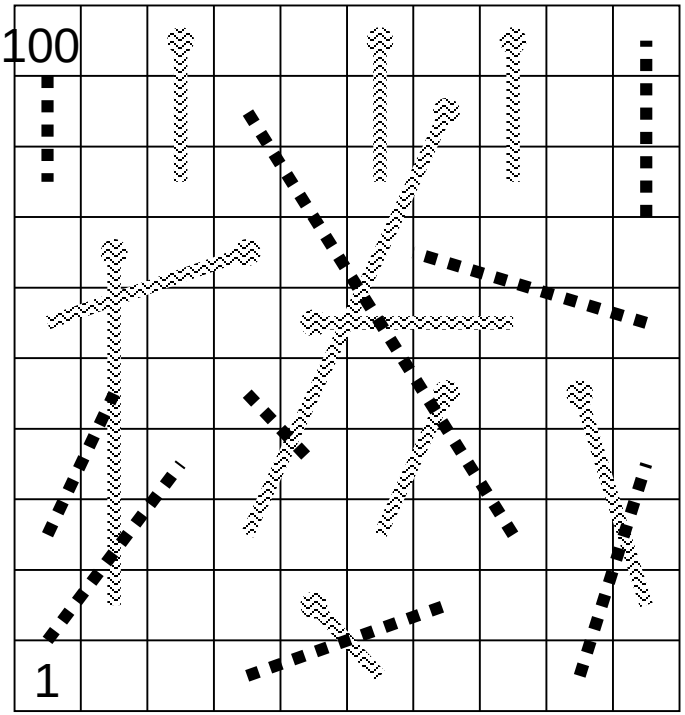

```
public class Course
{
  private Course prerequisite;
```

Snakes and Ladders

- A number of associations, including compositions, implemented in a simulated game

snake

ladder



Designing SnakesAndLadders

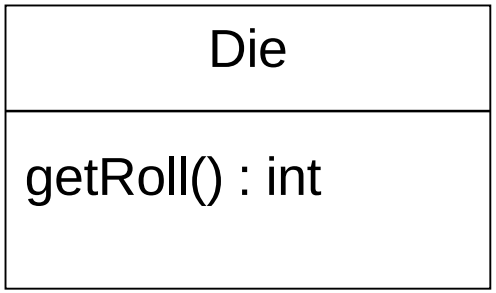
- Involves a number of new classes and also reuses the Die class from the last chapter
- We need to create objects of our classes and enable them to communicate with one another
- Deciding where responsibilities should lie
  - In OO design, put responsibilities in the ‘best’ class.
  - Sometimes the ‘best’ design is just a matter of opinion
  - Main guideline is to avoid having some objects that just contain data, and others that do all the process flow management
  - Better if data and processes are distributed through the classes in the place where they best fit

Game Objects

- The game, +
  - GameBoard
  - PlayerPiece
  - BoardSquare
  - Snake
  - Ladder
  - Die

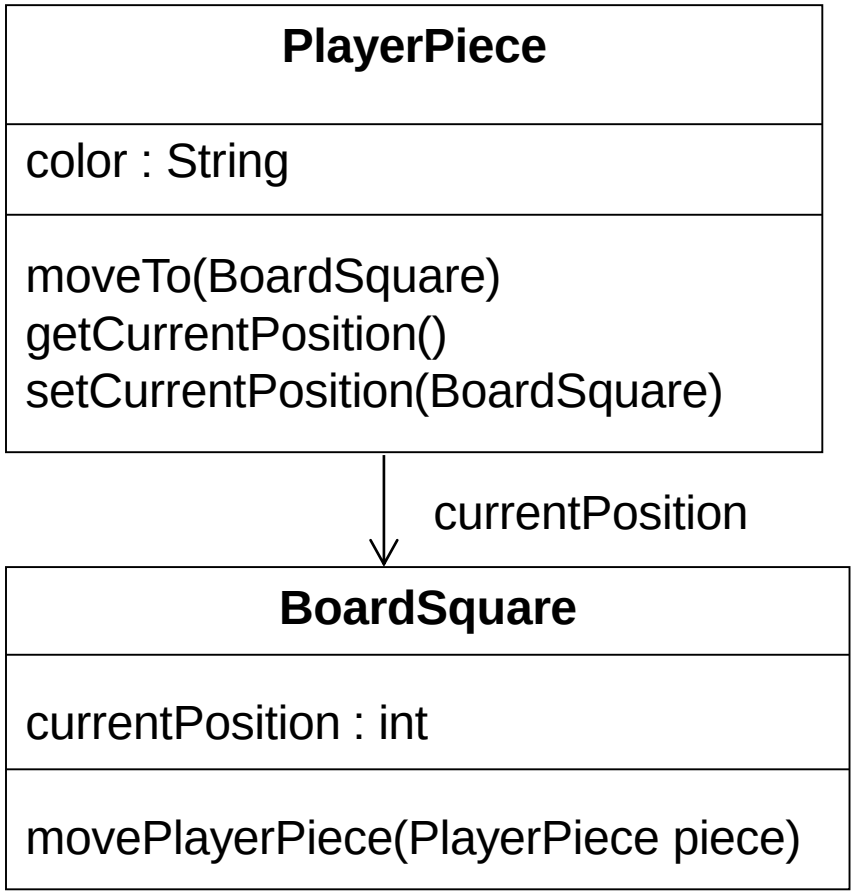
The Die Class

- Reuse from previous examples



PlayerPiece and BoardSquare

- PlayerPiece is responsible for knowing which BoardSquare it currently occupies
- Moving a PlayerPiece can be a two stage process
  - Initial move based on the throw of the die ('setCurrentPosition')
  - Then the possible move up a ladder or down a snake ('moveTo')



Snake and Ladder

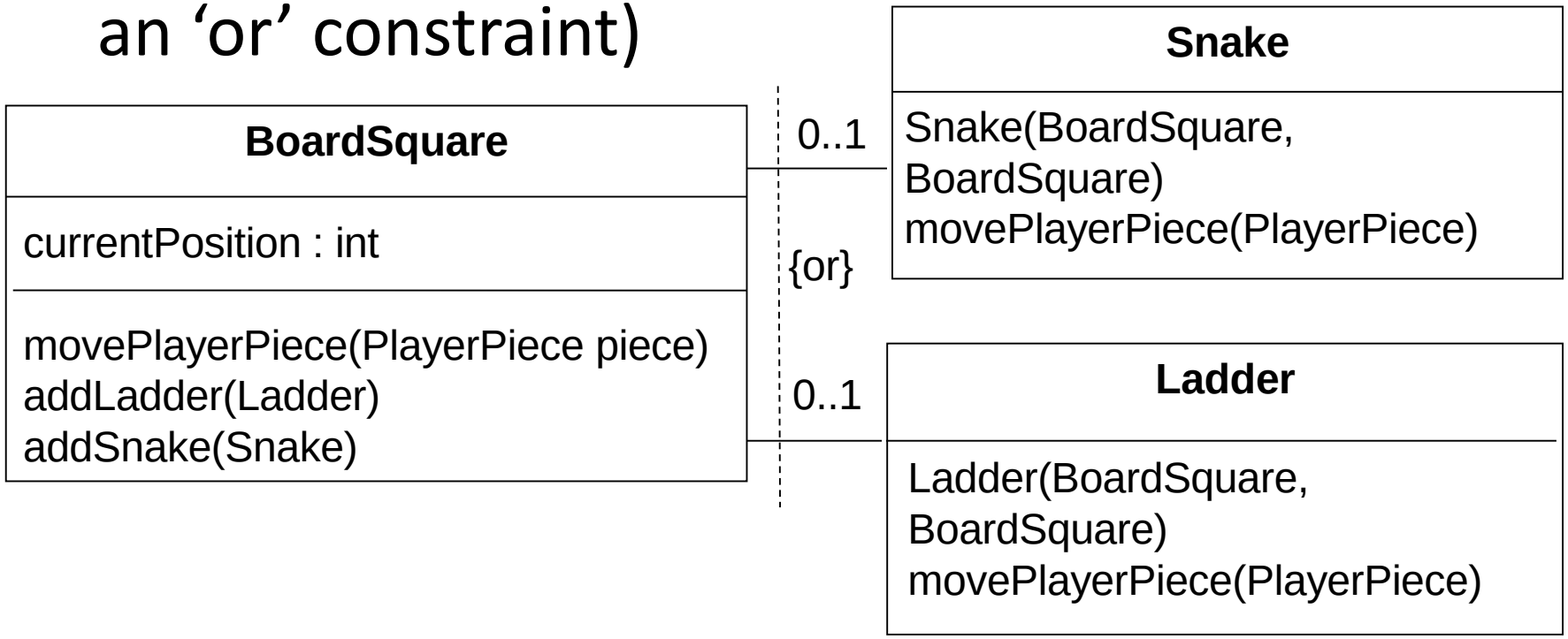
- Two separate classes (at least for now)
- The two BoardSquares that define the ends of the snake or ladder are passed to the constructor
  - as soon as a Snake or Ladder exists it knows where it fits on the board.

Snake
Snake(BoardSquare, BoardSquare) movePlayerPiece(PlayerPiece)

Ladder
Ladder(BoardSquare, BoardSquare) movePlayerPiece(PlayerPiece)

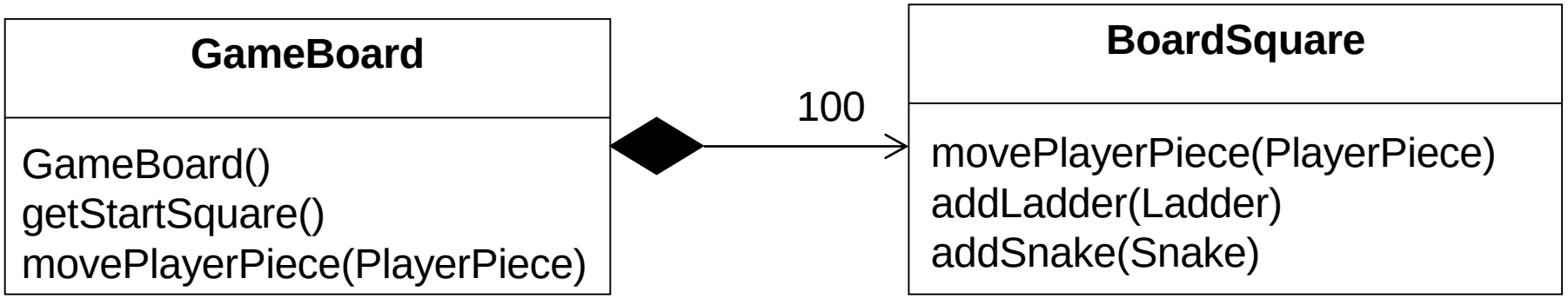
BoardSquare Associations

- Methods to add either a Snake or a Ladder
- Zero or one objects in each association (and an ‘or’ constraint)



GameBoard

- The GameBoard class represents the overall Snakes and Ladders board
- Comprises 100 BoardSquares
- Starts the process of moving a PlayerPiece with its 'movePlayerPiece' method.



Classes That Are Not 'public'

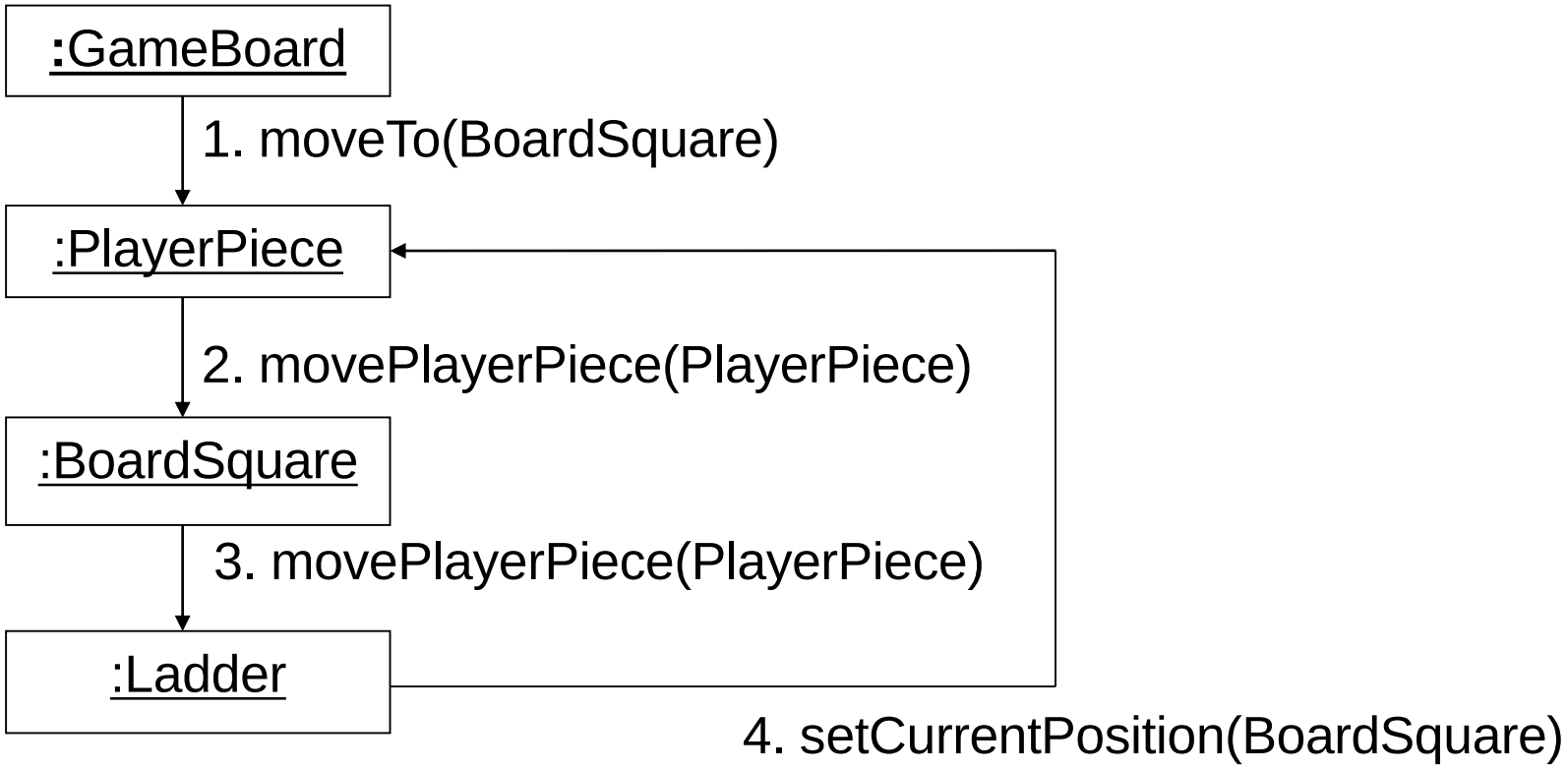
- Classes that are not declared public have 'package' visibility
  - Cannot be accessed from outside their package.
- Appropriate where they do not have any role beyond the specific programming context in which they are being used, e.g.
  - Die is 'public' - it may be used in any number of game programs
  - 'Snake' is not public - it is totally tied to the game of Snakes and Ladders
  - 'SnakesAndLadders' is public so that it can be accessed and played from other places
- If a class has package visibility, then it makes sense that its methods and constructors also have package (or private) visibility

Collaboration Diagrams

- These diagrams (also known as communication diagrams) show objects rather than classes
- An object is indicated by underlining the class name and preceding it with a colon
  - If the object has a specific name it can appear before the colon, but this can be omitted
- They help to explain how the various messages pass between the objects in sequence

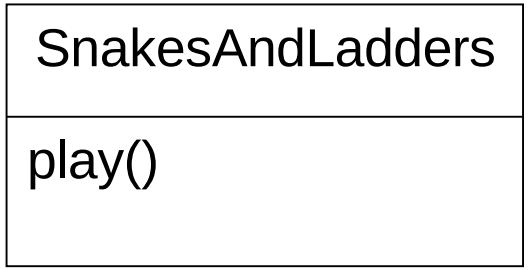
Example Collaboration Diagram

- A PlayerPiece is moved and lands on a BoardSquare containing the foot of a Ladder

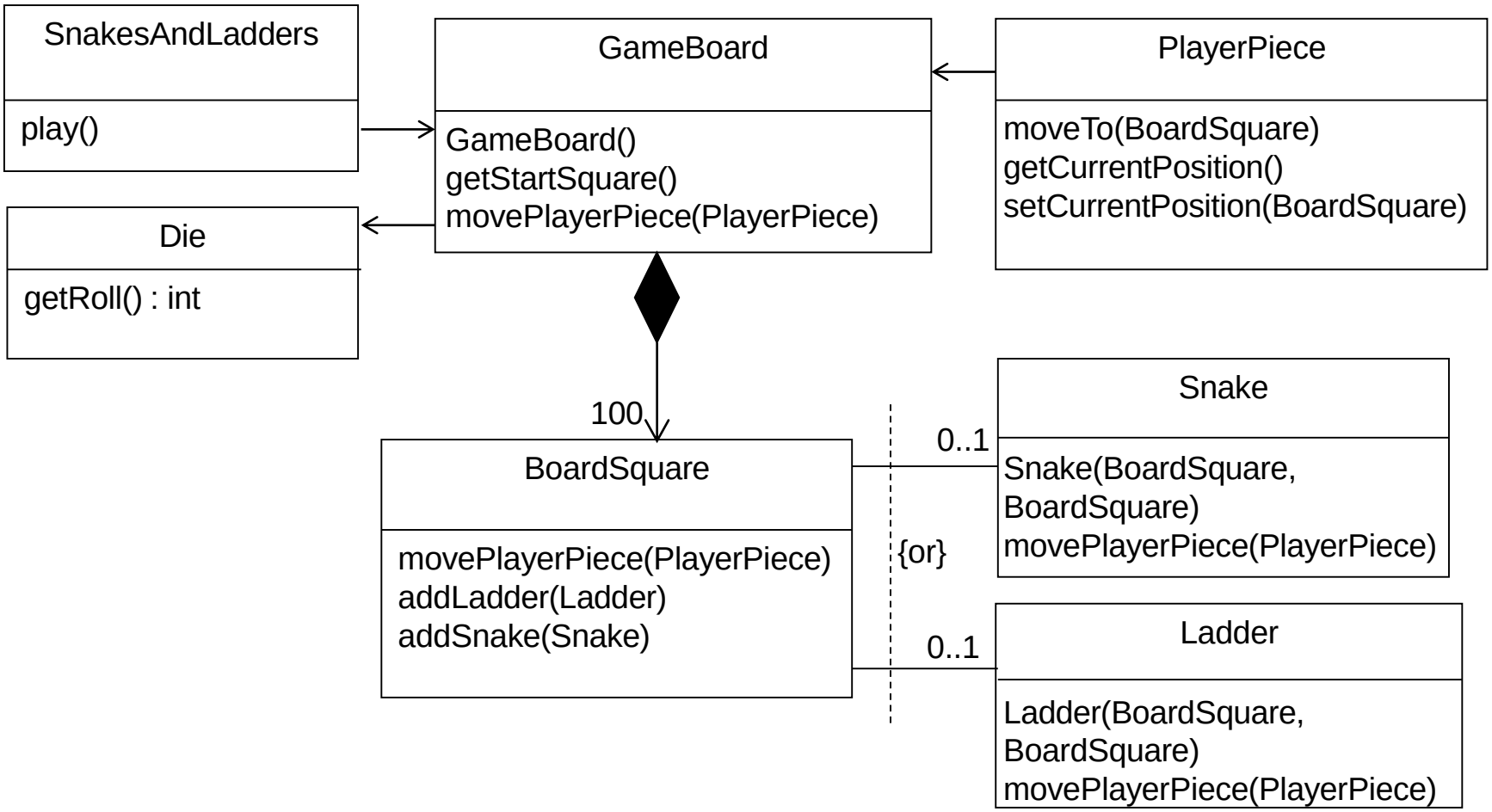


SnakesAndLadders Class

- Finally, the SnakesAndLadders class acts as the program entry point, and creates the GameBoard and the PlayerPieces
- It has a main method that creates an instance of the class and triggers the 'play' method
  - This handles the main game loop that iterates until the game is over



Complete Class Diagram



Making Associations

- Snake and Ladder constructors forge a relationship with two BoardSquares (at each end)

```
Snake(BoardSquare head, BoardSquare tail)
{
  setTail(tail);
  head.addSnake(this);
}
```

```
Ladder(BoardSquare top, BoardSquare foot)
{
  setTop(top);
  foot.addLadder(this);
}
```

Down Snakes, Up Ladders

- A Snake can move a piece to its tail

```
void movePlayerPiece(PlayerPiece counter)
{
    System.out.println("Down the snake to " + getTail().getPosition());
    counter.setCurrentPosition(getTail());
  }
}
```

- Ladders do something very similar

```
void movePlayerPiece(PlayerPiece counter)
{
    System.out.println("Up the ladder to " +  getTop().getPosition());
    counter.setCurrentPosition(getTop());
}
```

Moving a Piece

- The main responsibility of a BoardSquare is to move a PlayerPiece to 'this' BoardSquare, and possibly move it again along a Snake or a Ladder

```
public void movePlayerPiece(PlayerPiece counter)
{
    counter.setCurrentPosition(this);
    if (hasSnake()) {
      aSnake.movePlayerPiece(counter);
    }
    if (hasLadder()) {
      aLadder.movePlayerPiece(counter);
    }
  }
```

The GameBoard

- Constructor creates all the BoardSquares

```
squares = new BoardSquare[START_SQUARE + MAX_SQUARES];
for (int i = START_SQUARE; i <= MAX_SQUARES; i++)
{
    squares[i] = new BoardSquare(i);
}
```

- Then adds all the snakes and ladders to the chosen squares

The Main Game Loop

- The SnakesAndLadders class simulates the playing of the game
  - Creates a PlayerPiece
  - Moves it in a loop until it reaches square 100

```
public void play() {
   PlayerPiece counter = new PlayerPiece("Red");
   counter.setCurrentPosition(board.getStartSquare());
   while(counter.getCurrentPosition().getPosition() < GameBoard.MAX_SQUARES)
   {
            board.movePlayerPiece(counter);
   }
   System.out.println(counter.getColor() + " counter finished on " +
     counter.getCurrentPosition().getPosition());
}
```

When to Create Objects?

- We can creating objects where they are declared as fields, or create them in a method

```
 // reference to the GameBoard
private GameBoard board;
// the constructor creates the Board
public SnakesAndLadders()
{
  board = new GameBoard();
}
```

```
// reference to the GameBoard
private GameBoard board = new GameBoard();
```

- Create the object separately:
  - When object associations might change over time
  - When objects being created have parameterized constructors
- In some composition relationships it may well make sense to create the object when it is declared, rather than in the constructor

Exercise 7.1

- Add another PlayerPiece to the snakes and ladders game, and indicate which counter reaches the finish first

Association, Aggregation or Composition?

- The most important characteristic is ownership
- In an association objects do not own each other, only communicate
- In aggregation, one object may own other objects but they may also have an independent lifetime and other associations
- In composition, the whole owns its parts, and their lifetimes are probably identical. It is unlikely that the parts have any relationships with other objects outside the composition

Aggregation Example

- The Course and Module classes
- A Module represents a particular subject being taught as part of a larger course
  - a course on Java for example might have many modules relating to different aspects of the language
- A module has three attributes:
  1. the name of the module
  2. a credit point rating for certification (e.g., 20 credit points for a full module or 10 credit points for a half module)
  3. an assessment method (e.g., 'practicum', 'test', etc.)

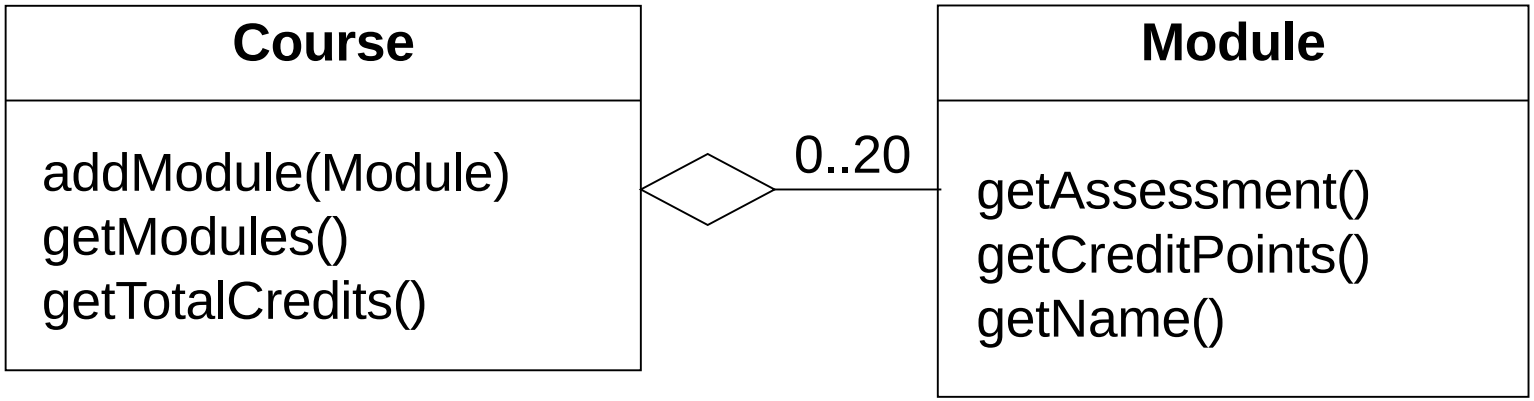
Module Fields and Constructor

```
public class Module
{
    private String name;
    private int creditPoints;
    private String assessment;
    public Module(String name, int points, String assess)
    {
        setName(name);
        setCreditPoints(points);
        setAssessment(assess);
    }
…
```

- The remainder of the class is getters and setters

Aggregation

- Modules are loosely coupled to courses
- Course objects are aggregations of zero or more Modules
- The maximum number of modules in a course is assumed to be 20



Course and Modules

```
public class Course {
// an array of modules
    private Module[] modules = new Module[20];
    private int moduleCount = 0;

// 'addModule' adds a parameter module to the array.
    public void addModule(Module newModule)
    {
        if(moduleCount < modules.length)
        {
            modules[moduleCount] = newModule;
            moduleCount++;
        }
        else
        {
            System.out.println("Cannot add more modules");
        }
    }
```

Exercise 7.2

- Create a Prospectus class that is an aggregation of many courses
- It should allow courses to be added and viewed
- Write a test class that creates a Prospectus object and tests its methods

Composition

- The next example shows how object composition can be used to create objects from components that are tightly bound together
- Real world objects are often clear examples of composition, because many objects are composed of smaller objects
- Electronic devices are very much of this type, and provide the context for this example

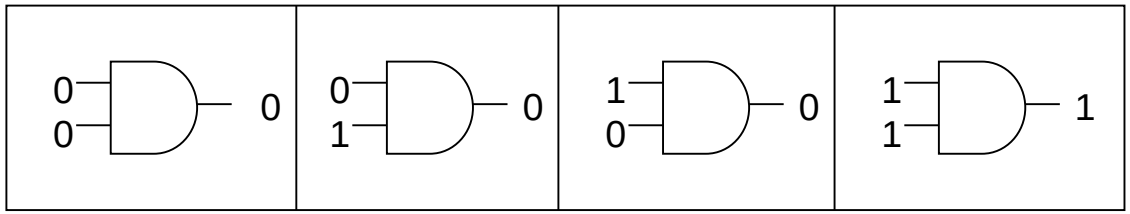
Electronic Gates

- A gate is a fundamental component of digital electronics
- The behavior of some types of gate will be very familiar to anyone who has used a programming language
- In Programming, Boolean operators can be used as part of the conditions used with selections

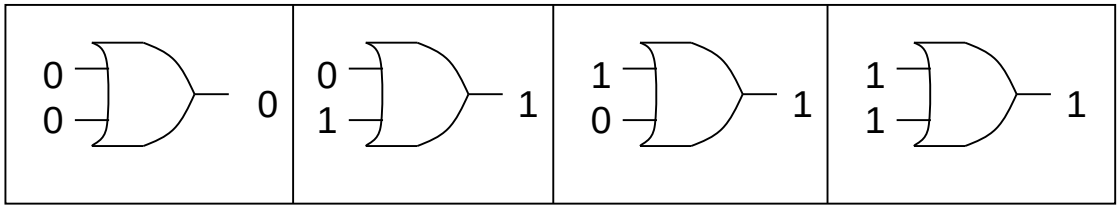
&&	AND	are both conditions true?
||	OR	is either of the conditions true?
!	NOT	is the condition false?

Gates as Boolean Components

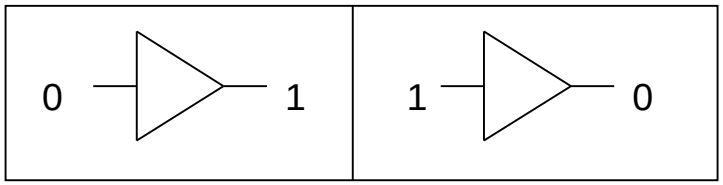
- AND gates



- OR gates



- NOT gates



Simple Functional Components

- AndGate example

```
public class AndGate
{
    public int getOutput(int input1, int input2)
    {
        if(input1 == 1 && input2 == 1)
        {
            return 1;
        }
        else
        {
              return 0;
        }
    }
}
```

Truth Table Code

- Example code to generate a truth table for an AndGate

```
AndGate andGate = new AndGate();
// output the column headings for the AND gate truth table
    System.out.println("Truth table for AND gate");
    System.out.println("\t0\t1");
// output the truth table fir the AND gate
   System.out.println("0\t" + andGate.getOutput(0,0) + "\t" + andGate.getOutput(0,1));
   System.out.println("1\t" + andGate.getOutput(1,0) + "\t" + andGate.getOutput(1,1));
```

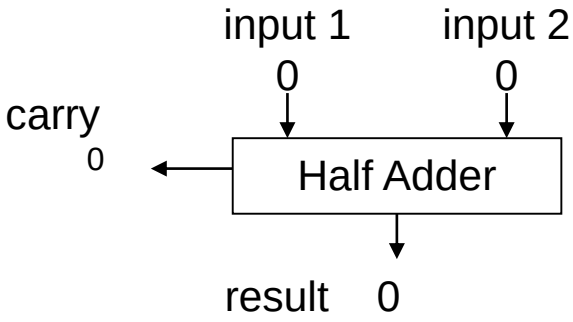
```
Truth table for AND gate
        0       1
0       0       0
1       0       1
```

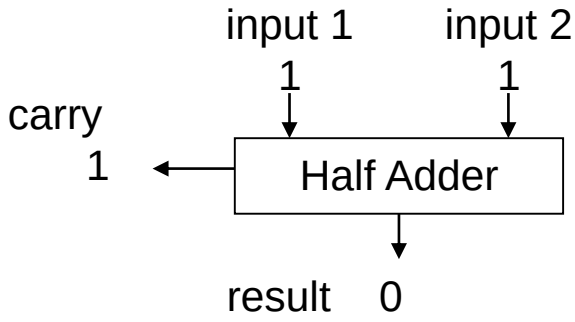
What Is a Half Adder?

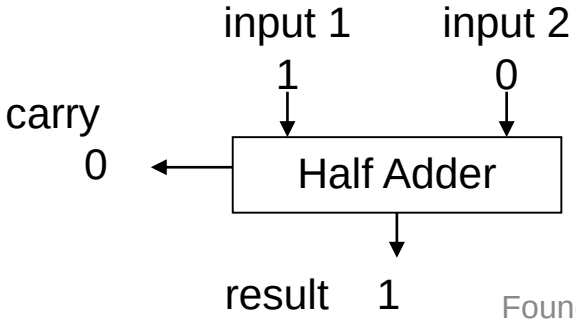
- One of the fundamental operations of a computer is to perform arithmetic on binary numbers
- It can do this by using collections of gates put together in particular ways
- One component that we can build simply from gates is the ‘half adder’
  - Able to add two binary digits, producing a result and a carry

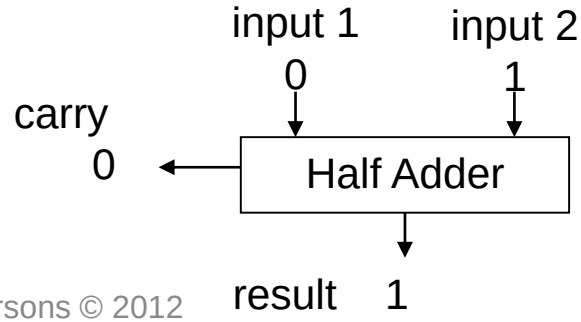
Half Adder Operations

- There are only four possible combinations of input bits to a half adder, and only three possible results



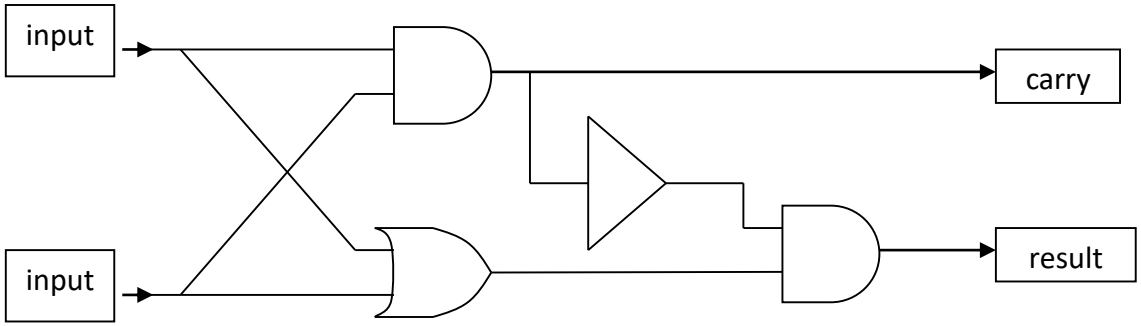




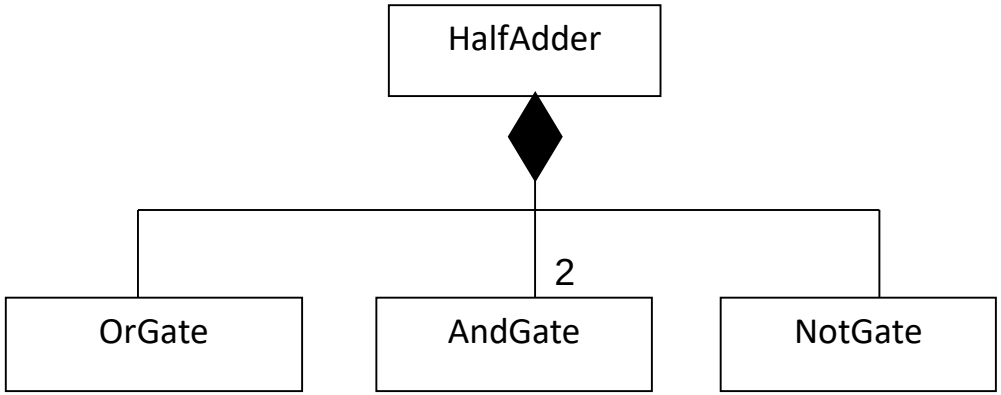


Half Adder

- As a circuit of gates



As a UML diagram of classes



Half Adder Fields

- A composition of gates
- Fields for input and output

```
public class HalfAdder
{
    AndGate carryAndGate = new AndGate();
    AndGate resultAndGate = new AndGate();
    OrGate orGate = new OrGate();
    NotGate notGate = new NotGate();
// inputs to the half adder
    private int input1;
    private int input2;
// outputs from the half adder
    private int result;
    private int carry;
```

Half Adder Methods

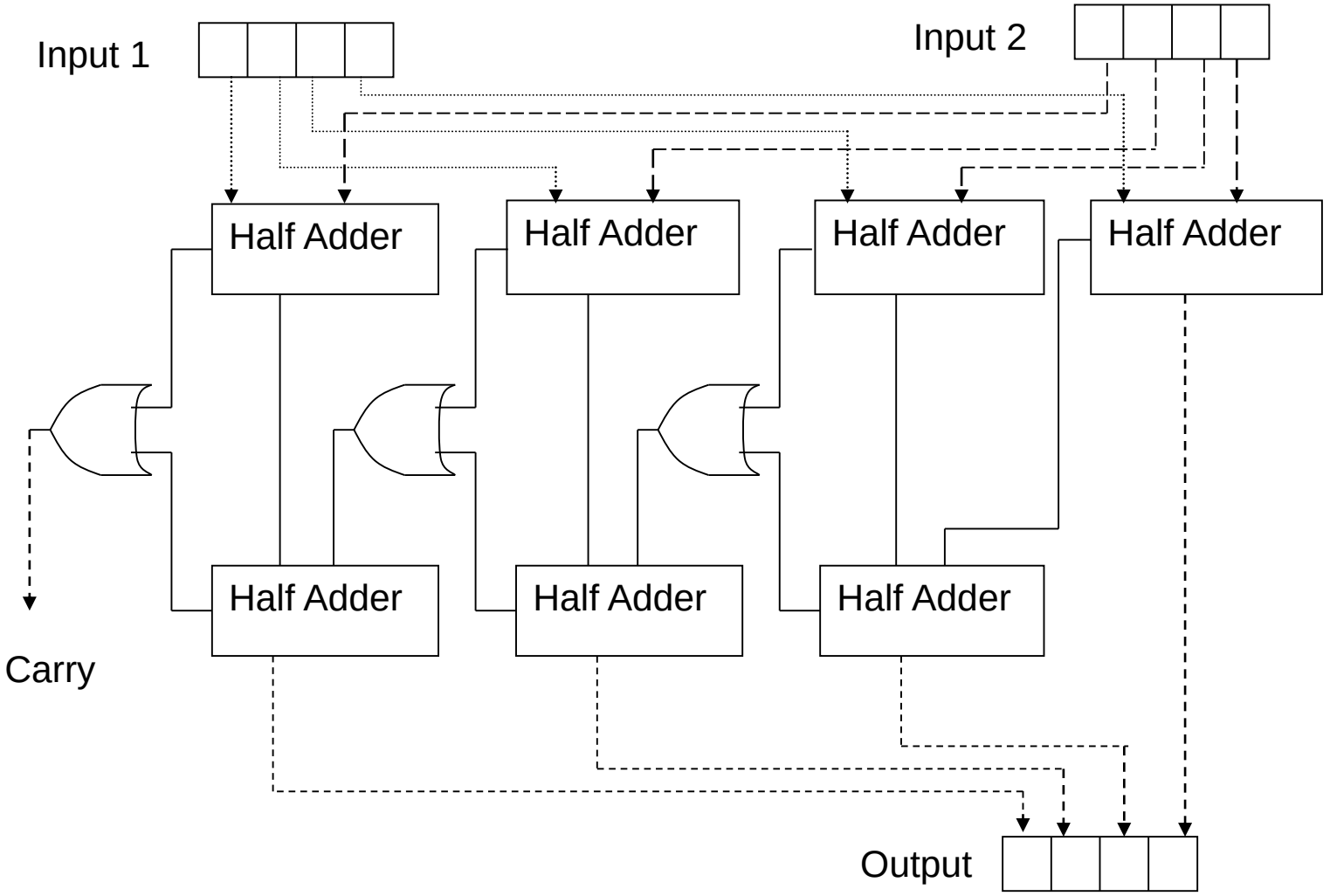
- Wiring the gates together

```
// set the values of the input bits
public void setInput(int in1, int in2)
{
    input1 = in1;
    input2 = in2;
// get the carry value
    carry = carryAndGate.getOutput(input1, input2);
// get the result value
    result = resultAndGate.getOutput
          (orGate.getOutput(input1, input2), notGate.getOutput(carry));
}
```

Exercise 7.3

- Using the existing HalfAdder and OrGate classes, write a FullAdder class that simulates a 4-bit full adder
  - See next slide (adapted from 'Illustrating Computers' by Day and Alcock, 1982)
- Test the FullAdder by using it to add various combinations of four-bit numbers

Exercise 7.3 – Full Adder



Summary

- Creating larger programs based on different objects communicating with one another
- Different ways that objects can work together
  - association: where independent objects talk to each other
  - aggregation: where an object is made up of other objects that can vary
  - composition: a very strong form of aggregation where the component objects are fixed