Photo by Marvin Meyer on Unsplash

Lab 3

Music · Pacman animation · Printing Sequences ·
Odds · Counting · Pacman maze

LüCE Lugano Computing Education Research Lab

Copy Your Lab 2 Toolbelt

In Lab 2 you wrote quite a few methods for your `Toolbelt` class. The `Toolbelt` class included in the Lab 3 starter repository is empty. Please **copy the contents of your Lab 2 `Toolbelt`** class into the Lab 3 `Toolbelt` class, so that you can continue to use the methods you develop (and add new ones you might need in the future).

A. Music with JTamaro

Task A1

JTamaro provides you with a method to play a sequence of notes.

```
void playNotes(Sequence<Note> notes, Instrument instrument)
```

Names for the notes using the Scientific Pitch Notation are defined inside the `jtamaro.en.music.Notes` class.

For example, the middle C is represented as `C4`. Sharp (♯) and flat (♭) are represented by putting a `S` or `F` respectively after the note name: for example, C♯ is represented as `CS4`.

Class:	`Music`
Task:	Let's familiarize ourselves with the JTamaro music API. In this task we will simply play the C major scale (ascending and descending) with an acoustic grand piano. To do that, first implement a parameter-less method `cMajorScale` that returns a sequence of notes from C4 to B4. The "alphabet" of notes ranges from A to G, and then "wraps". That means that the note following C4 should be a D4, and so on. After G4, A4 follows. Then, define a parameter-less method `cMajorAscendingDescending` that returns a sequence that contains the notes of the `cMajorScale` sequence, `C5` and the notes of the `cMajorScale` sequence in **reverse** order. The `of`, `concat` and `reverse` methods for `Sequence`s are helpful to solve this task.
Run in JShell:	`playNotes(Music.cMajorScale(), Instrument.ACOUSTIC_GRAND_PIANO)`
Output:	Listen to the first part of c-major.mp3 and compare
Run in JShell:	`playNotes(Music.cMajorAscendingDescending(), Instrument.ACOUSTIC_GRAND_PIANO)`
Output:	Listen to c-major.mp3 and compare

Task A2

The Swiss Railways (SBB-CFF-FSS) use three versions of the jingle for travel announcements. The three jingles are made with the notes of the three acronyms:

- E(s) – B – B for the German SBB (*Schweizerische Bundesbahnen*)
- C – F – F for the French CFF (*Chemins de fer fédéraux suisses*)
- F – E(s) – E(s) for the Italian FFS (*Ferrovie federali svizzere*)

The jingle played depends on which canton (or country for international travels) the station or train is located in. For example, if you were to take the EC35 train that travels from Genève to Venezia, you would first hear the CFF jingle in Genève, then the SBB jingle in Brig and once the train crosses the Italian border, you'd finally hear the FFS one.

You can find more information on the topic by visiting these articles on the SBB website.

We will use the Vibraphone as our instrument to match the original sound.

Class:	`Music`
Task:	Implement the `sbb` method to return a sequence containing the following notes: E♭4, B♭4 and B♭4. Implement the `cff` method to return a sequence containing the following notes: C5, F4 and F4. Implement the `ffs` method to return a sequence containing the following notes: F4, F4 and E♭4. Implement the `allJingles` method to return a sequence containing the notes of the `sbb`, `cff` and `ffs` jingles in sequence. **Important**: use the `cons`, `empty`, `of` and `concat` methods appropriately to create the different sequences.
Run in JShell:	`playNotes(Music.sbb(), Instrument.VIBRAPHONE)`
Output:	Listen to sbb.mp3 and compare
Run in JShell:	`playNotes(Music.cff(), Instrument.VIBRAPHONE)`
Output:	Listen to cff.mp3 and compare
Run in JShell:	`playNotes(Music.ffs(), Instrument.VIBRAPHONE)`
Output:	Listen to ffs.mp3 and compare
Run in JShell:	`playNotes(Music.allJingles(), Instrument.VIBRAPHONE)`
Output:	Listen and compare to the previously linked mp3 files

B. Animated Pacman

So far, we have shown individual graphics by using the method `show`:

```
void show(Graphic graphic)
```

JTamaro also provides the `showFilmStrip` method to show a sequence of graphics as a film strip.

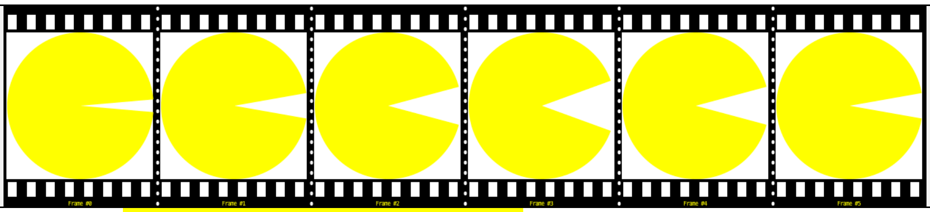
```
void showFilmStrip(Sequence<Graphic> frames, int width, int height)
```

The method has three parameters: a sequence of graphics (one graphic for each frame of the film strip), the width of a frame, and the height of a frame. To get a decent result, the graphics in the sequence should be no bigger than the frame.

Finally, JTamaro also provides the animate method that shows a sequence of graphics as a looped animation.

```
void animate(Sequence<Graphic> frames)
```

Task B1

Class:	`PacmanSprite`
Task:	Implement the `pacman` and the `pacmans` method. You probably didn't move your old `pacman` method from `lab-01` into your toolbelt. If you did, you may now use it. If you did not copy it in the past, either copy it into your toolbelt now, or write the method again in the `PacmanSprite.java` file. The `pacmans` method should return a sequence of graphics. Each graphic should show a pacman with the given `diameter` and with a different `mouthAngle`. The sequence should contain 6 frames with pacman widening its mouth from a `mouthAngle` of `10` to `40`. Remember to include frames for both the widening and narrowing of the mouth angle, otherwise the animation will not look good. **Assert** that the `diameter` and `mouthAngle` are acceptable.
Run in JShell:	`showFilmStrip(PacmanSprite.pacmans(100), 100, 100)`
Output:	
Run in JShell:	`animate(PacmanSprite.pacmans(100))`
Output:	Animation of pacman opening and closing its mouth
	Do you feel the pain of having to write the almost same code six times to produce six pacmans?

C. Printing Sequences

JTamaro has two methods that print the content of a sequence:

```
<T> void print(Sequence<T> sequence)

<T> void println(Sequence<T> sequence)
```

The former prints the elements on the same line, the latter prints each element on a separate line.

These methods work well with sequences of integers, doubles, booleans, characters, and strings. While you can use them to print sequences of colors, or graphics, or other things, the printed output will not actually show you the color or the graphic, but just some text. If you want to look at a sequence of graphics, use `showFilmStrip` or `animate` instead.

Task C1

Class:	`Ranges`
Task:	Implement the `degreeAngles` method to return a sequence of angles: from 0 degrees to 359 degrees with a given step value. **Important:** Use the `range` method.
Run in JShell:	`println(Ranges.degreeAngles(1))`
Output:	0 1 2 … 359
Run in JShell:	`println(Ranges.degreeAngles(3))`
Output:	0 3 6 … 357
	Do you feel the joy of producing this sequence with such a short piece of code?

D. Odds[1]

We want to model the likelihood, or *odds*, of an event. For example, if we were to roll a fair six-sided die, the probability of rolling a 6 would be "one out of six" (one favorable case, five unfavorable cases, six cases in total).

Another way to represent this would be "five-to-one", or "5/1" in short. This notation conveys the fact that there are 5 ways to not roll the desired number and 1 way to roll it. Finally, another way would be to use a percentage to represent the event likelihood (16.67% in this case, or 0.16666666...).

Task D1

Class:	`Odds (You need to create it)`
Task:	Implement a record class in the file named `Odds.java`. This record class must have two fields named `will` and `wont`, both of type `int`, to keep track of the number of *favorable* cases in which the event ***will*** happen and the number of *unfavorable* cases in the event ***won't*** happen, respectively. **Note:** throughout this exercise, never attempt to "simplify" the fraction representing the odds (i.e., it is perfectly fine to model an event that has 3 favorable cases and 3 unfavorable ones; no need to reduce it to 1 and 1).
Run in JShell:	`new Odds(1, 5)`
Output:	`==> Odds[will=1, wont=5]`

Task D2

Class:	`OddsUtils`
Task:	In the file `OddsUtils.java`, define three parameter-less methods: • `rollingSix()` that returns an instance of `Odds` representing the probability (1 out of 6) to roll a six using a fair six-sided die • `tossingHeads()` that returns an instance of `Odds` representing the probability (1 out of 2) to toss heads using a fair coin • `drawingAce()` that returns an instance of `Odds` representing the probability (4 out of 52) to draw an ace from a deck of cards
Run in JShell:	`OddsUtils.rollingSix()`
Output:	`==> Odds[will=1, wont=5]`
Run in JShell:	`OddsUtils.tossingHeads()`
Output:	`==> Odds[will=1, wont=1]`
Run in JShell:	`OddsUtils.drawingAce()`
Output:	`==> Odds[will=4, wont=48]`

---

[1] This exercise is inspired by an assignment by Juha Sorva.

Task D3

Class:	`OddsUtils`
Task:	Implement a method named `probability`, which returns as a `double` the probability of a given `Odds` instance. The probability is computed as the ratio of favorable cases (`will`) over the total number of cases (favorable and unfavorable). To compute the total number of cases, define and then call an auxiliary method named `cases` that returns as an `int` the appropriate value. This method will come in handy later. **Hint**: if you get `0` as a result, you might be preforming a division between two integer values (note that both `will` and `wont` are of type `int`). There are many ways to make Java perform a division with floating point numbers (e.g., `double` values). One simple way is to first multiply the dividend by `1.0` (the `.0` part of the literal is important as it is what specifies that the literal is a `double` and not an `int`).
Run in JShell:	`OddsUtils.probability(OddsUtils.rollingSix())`
Output:	`==> 0.16666666666666666`
Run in JShell:	`OddsUtils.probability(OddsUtils.tossingHeads())`
Output:	`==> 0.5`
Task:	Implement a method named `fractional` method, such that, given an `Odds` instance, it returns a string representation of it in the format `"wont/will"` (to be read as *wont*-to-*will*, as in “five-to-one”)
Run in JShell:	`OddsUtils.fractional(OddsUtils.rollingSix())`
Output:	`==> "5/1"`
Run in JShell:	`OddsUtils.fractional(OddsUtils.drawingAce())`
Output:	`==> "48/4"`
Task:	Implement the `decimal` method now, such that, given an `Odds` instance, it returns the *reciprocal* (i.e., the inverse) of its probability. This value describes the odds in “one-in-*how-many*” terms. For example, drawing an ace out of a deck of cards has a one-in-*thirteen* (cases) chance of happening.
Run in JShell:	`OddsUtils.decimal(OddsUtils.drawingAce())`
Output:	`==> 13.0`
Run in JShell:	`OddsUtils.decimal(OddsUtils.rollingSix())`
Output:	`==> 6.0`

Task D4

Class:	`OddsUtils`
Task:	In a betting context, the return value of the `decimal` method you just implemented is the number that a bettor's investment multiplies by. For example, if the odds of an event such as Switzerland winning the next Eurovision Song Contest are five-to-two, the successful bettor will receive 3.5 times what they bet (they would get their money back plus 2.5 times that much extra). Implement the `winnings` method that computes, given an `Odds` instance and the amount invested (type `double`), the amount of winnings a successful bettor would get.
Run in JShell:	`OddsUtils.winnings(new Odds(2, 5), 20.0)`
Output:	`==> 70.0`

Task D5

Class:	`OddsUtils`
Task:	Implement a method, called `complement`, that computes the complementary event by inverting a given `Odds` instance. For example, the complementary event of rolling a six is "not rolling a six".
Run in JShell:	`OddsUtils.complement(OddsUtils.rollingSix())`
Output:	`==> Odds[will=5, wont=1]`
Run in JShell:	`OddsUtils.complement(OddsUtils.complement(OddsUtils.rollingSix()))`
Output:	`==> Odds[will=1, wont=5]`

Task D6

Class:	`OddsUtils`
Task:	Another curious way of representing the likelihood of an event, used by many North American betting agencies, is the “moneyline” format. **In case the event’s estimated probability is at most 50%,** its moneyline number is positive and is computed as `100 * wont / will`. For example, the moneyline number for 7-to-2 odds is 350, because $100 \times \frac{7}{2} = 350$. This positive number indicates that if you bet 100 monetary units and win, you profit 350 units in addition to getting your bet back. A fifty–fifty scenario (1-to-1 odds) has a moneyline number of 100. **In case the event’s estimated probability is over 50%,** its moneyline number is negative and is computed as `-100 * will / wont`. For example, the moneyline number for 1-to-5 odds is -500, because $-100 \times \frac{5}{1} = -500$. This negative number indicates that if you want to make a profit of 100 units, you have to place a bet of 500 units. Implement the `moneyline` method that, given an Odds instance, returns its `moneyline` value according to the two cases specified above.
Run in JShell:	`OddsUtils.moneyline(new Odds(2, 3))`
Output:	`==> 150.0`
Run in JShell:	`OddsUtils.moneyline(new Odds(3, 1))`
Output:	`==> -300.0`

Task D7

Class:	`OddsUtils`
Task:	Implement a `cases2` method which computes the number of possible cases for the combined outcome of two given events. For example, two rolls of a die give rise to 36 (6 * 6) possible cases. A roll of a die and a toss of a coin give rise to 12 (6 * 2) possible cases.
Run in JShell:	`OddsUtils.cases2(OddsUtils.rollingSix(), OddsUtils.rollingSix())`
Output:	`==> 36`
Run in JShell:	`OddsUtils.cases2(OddsUtils.rollingSix(), OddsUtils.tossingHeads())`
Output:	`==> 12`
Task:	Implement the `both` method which takes two `Odds` instances and returns an `Odds` instance representing the likelihood of *both* events happening together.

	The number of favorable cases in the combined event is the product of the favorable cases of the two individual events. The number of unfavorable cases can be computed by subtracting from the total number of cases (cf. `cases2`) the favorable ones.
Run in JShell:	`OddsUtils.both(OddsUtils.rollingSix(), OddsUtils.rollingSix())`
Output:	`==> Odds[will=1, wont=35]`
Run in JShell:	`OddsUtils.both(OddsUtils.tossingHeads(), OddsUtils.tossingHeads())`
Output:	`==> Odds[wont=1, will=3]`
Task:	Implement the `either` method which takes two `Odds` instances and returns an `Odds` instance representing the likelihood of *either* of the events happening (the first one, the second one, or both). The number of unfavorable cases in the combined event is the product of the unfavorable cases of the two individual events. The number of favorable cases can be computed by subtracting from the total number of cases (cf. `cases2`) the unfavorable ones.
Run in JShell:	`OddsUtils.either(OddsUtils.tossingHeads(), OddsUtils.tossingHeads())`
Output:	`==> Odds[will=3, wont=1]`
Run in JShell:	`OddsUtils.either(OddsUtils.rollingSix(), OddsUtils.rollingSix())`
Output:	`==> Odds[will=11, wont=25]`

Task D8

Class:	`OddsUtils`
Task:	Implement an `eventHappens` method that returns a `boolean` value simulating whether an event with certain `Odds` happens, using a randomly generated value. You can call the `Math.random()` method to generate a random floating-point number in the range [0, 1). **Hint:** implement this method by comparing the random value with the one returned by the `probability` method.
Run in JShell:	`OddsUtils.eventHappens(OddsUtils.tossingHeads())`
Output:	`false`
Run in JShell:	`OddsUtils.eventHappens(OddsUtils.tossingHeads())`
Output:	`false`
Run in JShell:	`OddsUtils.eventHappens(OddsUtils.tossingHeads())`
Output:	`true`

E. Counting

Imagine a planet where inhabitants were prohibited from talking to themselves. You could talk to every other inhabitant on the planet, but you would be **punished** if you ever dared to talk **to yourself**.

Some arcane programming languages have such a strange limitation: A function can call every other function, with one exception: it cannot call itself.
Why? There is a technical reason: If a function cannot be called while it is already executing, then the runtime system does not need to maintain a call stack.
In modern languages, a function can call every function, including itself.

While it may be somewhat rare that humans talk to themselves, in programming languages, it's quite common that functions call themselves. This provides a powerful and general way of **repeating a computation**. Functions that call themselves are called "recursive". You wrote recursive functions in Racket *SL.

Task E1

Class:	`Recurse`
Task:	Using recursion, implement the `length` method to return the length of the given sequence of colors. Remember key concepts when developing a recursive function: termination condition, base case, and recursive case. **Important**: The length method has a type parameter <T> that allows it to work with any different Sequence type instances (e.g, Sequence<Integer>, Sequence<String>, Sequence<Graphic>, ...). **Important:** Use the `isEmpty` and `rest` methods.
Run in JShell:	`Recurse.length(of())`
Output:	`0`
Run in JShell:	`Recurse.length(empty())`
Output:	`0`
Run in JShell:	`Recurse.length(cons(true, empty()))`
Output:	`1`
Run in JShell:	`Recurse.length(of('a', 'b','c'))`
Output:	`3`
Run in JShell:	`Recurse.length(range(0, 360, 36))`
Output:	`10`
Run in JShell:	`Recurse.length(replicate(4, 10))`
Output:	`10`
Run in JShell:	`Recurse.length(PacmanSprite.pacmans(100))`
Output:	`6`

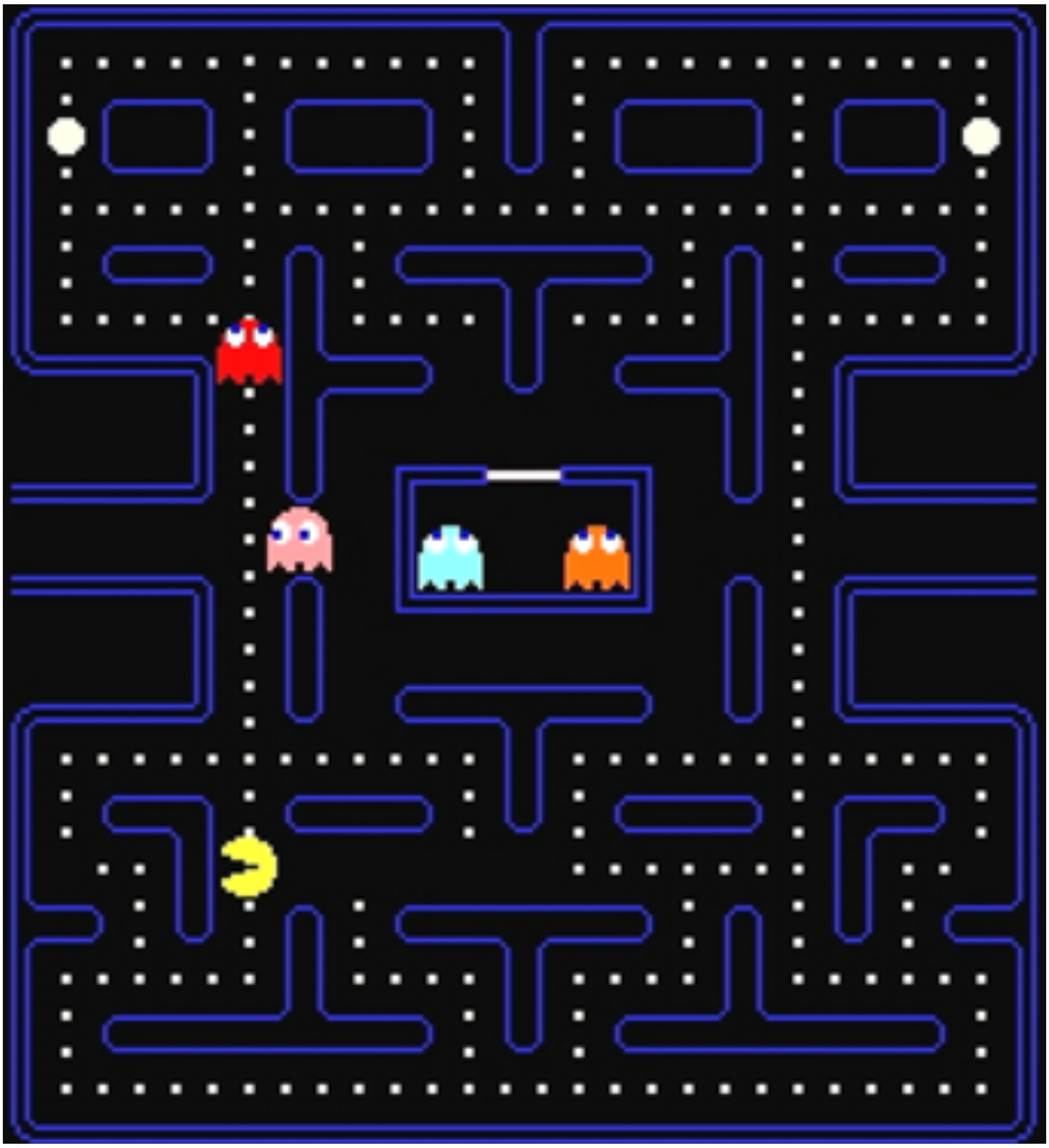
F. Pacman Maze

We will now start creating graphics of assets of a game. We will re–use the code that you write here in a future lab so that you may build your own working Pacman game.

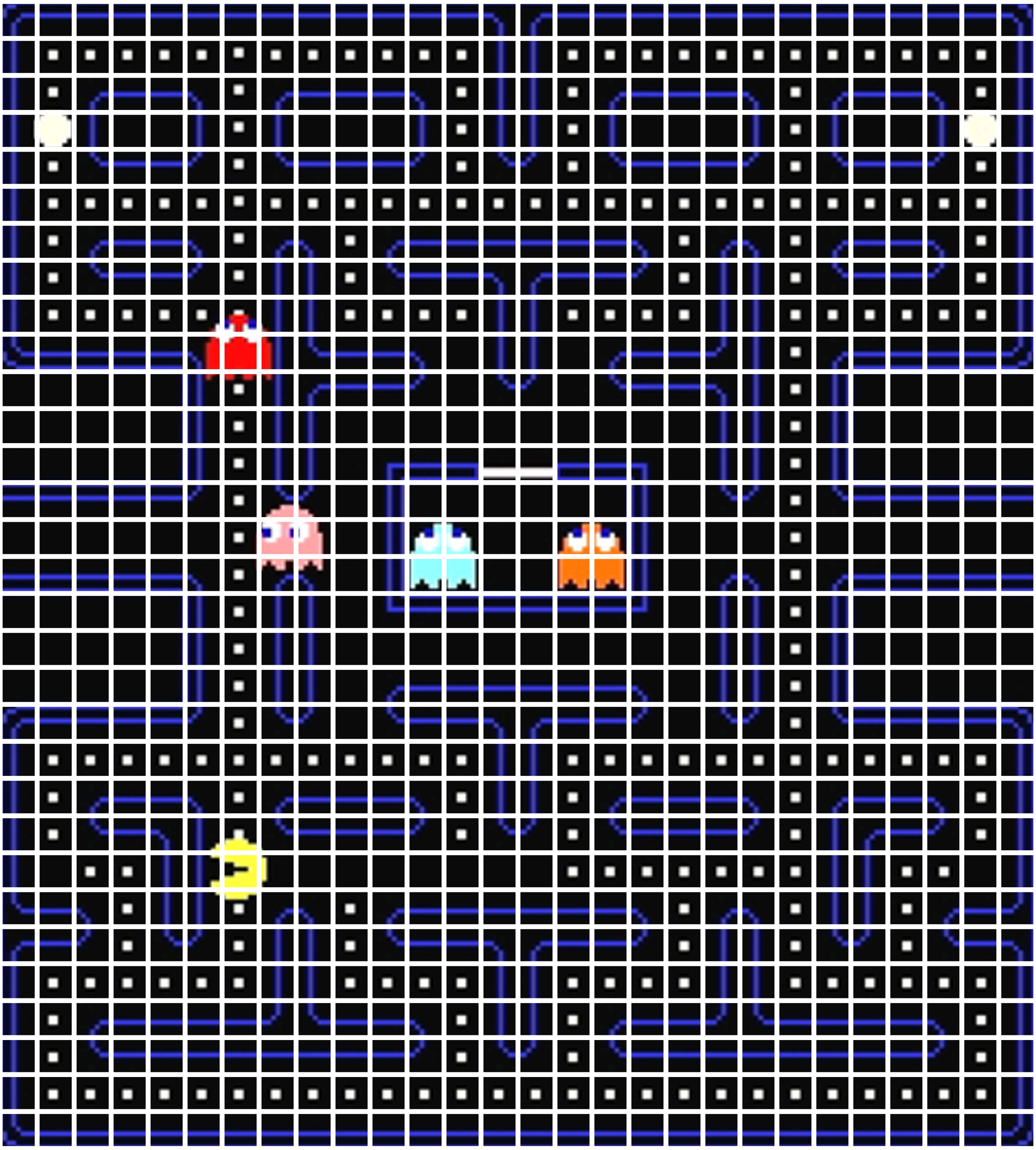
You have already created a pacman sprite during Task B1, let's now prepare the graphics to build the maze.

The maze of a pacman game seems to follow a rather regular structure.

Without looking at the next page, can you decompose this pacman maze into smaller graphics?



Here is a possible decomposition:

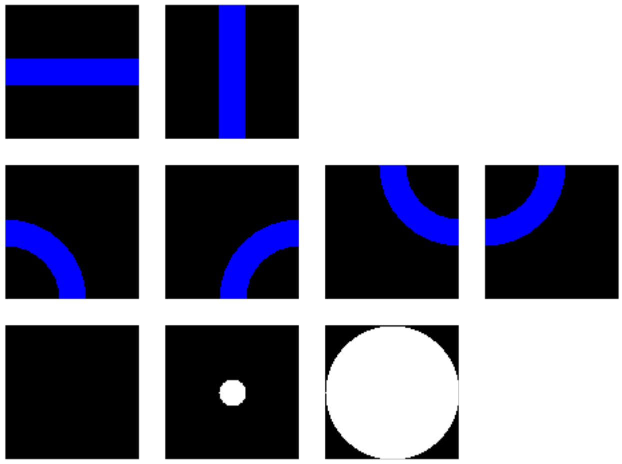


We could compose the maze from a lot of square-shaped tiles.

How many kinds of tiles do we need? (Ignore the pacman and ghosts.)

Figure that out **before looking at the next page**.

If we ignore small differences, we can see six kinds of wall tiles (2 straight walls and 4 corners) and three kinds of floor tiles (just black, black with a small white dot, and black with a large white pill).



Let's write three methods to produce these kinds of tiles.

Task F1

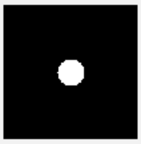
Class:	`PacmanMaze`
Task:	Implement the `straight` method to return a graphic of a horizontal or vertical straight tile. The line should be centered, and 1/5 as thick as the tile's size. **Assert** that the `size` is acceptable. **Use** your toolbelt.
Run in JShell:	`show(PacmanMaze.straight(100, true))`
Output:	
Run in JShell:	`show(PacmanMaze.straight(100, false))`
Output:	

Task F2

Class:	`PacmanMaze`
Task:	Implement the `corner` method to return a graphic of a corner tile, rotated by the given number of degrees. **Assert** that the `rotation` is a multiple of 90. To give a helpful error message when the assertion is violated, provide a string: `assert condition : string;` **Assert** that the `size` is acceptable. **Use** your toolbelt.
Run in JShell:	`show(PacmanMaze.corner(100, 0))`
Output:	
Run in JShell:	`show(PacmanMaze.corner(100, 90))`
Output:	
Run in JShell:	`show(PacmanMaze.corner(100, 180))`
Output:	
Run in JShell:	`show(PacmanMaze.corner(100, 270))`
Output:	
Run in JShell:	`show(PacmanMaze.corner(100, 45))`
Output:	Must throw an `AssertionError`. Expected output in JShell should resemble the following text (numbers in the `at` lines may be different for you):

```
|  Exception java.lang.AssertionError
|        at PacmanMaze.corner (PacmanMaze.java:21)
|        at (#26:1)
```

Task F3

Class:	`PacmanMaze`
Task:	Implement the `floor` method to return a graphic of a floor tile, either empty, with a dot, or with a pill. The parameter `dot` determines whether the tile should contain a dot. The parameter `pill` determines whether the tile should contain a pill. **Assert** that we are not asked to produce a dot and a pill at the same time. **Assert** that the `size` is acceptable. **Use** your toolbelt.
Run in JShell:	`show(PacmanMaze.floor(100, false, false))`
Output:	
Run in JShell:	`show(PacmanMaze.floor(100, true, false))`
Output:	
Run in JShell:	`show(PacmanMaze.floor(100, false, true))`
Output:	
Run in JShell:	`show(PacmanMaze.floor(100, true, true))`
Output:	Must throw an `AssertionError`. Expected output in JShell should resemble the following text (numbers in the `at` lines may be different for you):

```
|  Exception java.lang.AssertionError
|        at PacmanMaze.floor (PacmanMaze.java:48)
|        at (#21:1)
```

Task F4

Class:	`PacmanMaze`
Task:	Implement the `demoMaze` method to return a graphic of a bunch of tiles composed into rows and columns. Your maze must consist of at 3-by-3 tiles with the outer tiles representing the walls and the floor tile in the center containing the small dot. **Assert** that the `tileSize` is acceptable. **Use** your toolbelt (e.g., `above3` and `beside3`).
Run in JShell:	`show(PacmanMaze.demoMaze(100))`
Output: