Designing with methods that call other methods

- Design a Java program to measure reaction time of users responding to true and false "facts".
  - Ask the user about a fact: "Is it true that the BE is a 4 Year degree?"
  - Measure the time they took
  - Print out how much time.

- Need a class
  - what name?
- Need a method
  - what name?
  - what parameters?
  - what actions?

ReactionTimeMeasurer

```
/** Measures reaction times for responding to true-false statements */
public class ReactionTimeMeasurer {

/** Measure and report the time taken to react to a question */
    public void measureReactionTime() {
        .  // find out the current time and remember it
           // ask the question and wait for answer
        .  // find out (and remember) the current time
           // print the difference between the two times
    }
}
```

Write the method body in comments first,

(to plan the method without worrying about syntax)

Work out what information needs to be stored (ie, variables)

ReactionTimeMeasurer

```
    /** Measure and report the time taken to react to a question */
    public void measureReactionTime() {
        long startTime = System.currentTimeMillis();
        UI.askString("Is it true that the sky is blue?");
        long endTime = System.currentTimeMillis();
        UI.printf("Reaction time = %d milliseconds \n",  (endTime - startTime) );
    }
}
```

Returns a very big integer
⇒ long
(milliseconds since 1/1/1970

Just asking one question is not enough for an experiment.

➔ need to ask a sequence of questions.

Multiple questions, the bad way

```
/** Measure and report the time taken to react to a question */
public void measureReactionTime(){
    long startTime = System.currentTimeMillis();
    UI.askString( "Is it true that John Quay was the Prime Minister");
    long endTime = System.currentTimeMillis();
    UI.printf("You took %d milliseconds \n",  (endTime - startTime) );

    startTime = System.currentTimeMillis();
    UI.askString( "Is it true that 6 x 4 = 23");
    endTime = System.currentTimeMillis();
    UI.printf("You took %d milliseconds \n",  (endTime - startTime) );

    startTime = System.currentTimeMillis();
    UI.askString( "Is it true that summer is warmer than winter");
    endTime = System.currentTimeMillis();
    UI.printf("You took %d milliseconds \n",  (endTime - startTime) );

    startTime = System.currentTimeMillis();
    UI.askString( "Is it true that Wellington's population > 1,000,000");
    endTime = System.currentTimeMillis();
    UI.printf("You took %d milliseconds \n",  (endTime - startTime) );
}
```

Lots of repetition.
But not exact repetition.
How can we improve it?

Good design with methods

- Key design principle:
  - Wrap up repeated sections of code into a separate method,
  - Call the method several times:

```
public void measureReactionTime ( ) {
    this.measureQuestion( "John Quay was the Prime Minister");
    this.measureQuestion( "6 x 4 = 23");
    this.measureQuestion( “Summer is warmer than winter");
    this.measureQuestion( "Wellington’s population > 1,000,000 ");
}

public void measureQuestion ( String fact ) {
    long startTime = System.currentTimeMillis();
    UI.askString("Is it true that " + fact . );
    long endTime = System.currentTimeMillis();
    UI.printf("You took %d milliseconds \n",  (endTime - startTime) );
}
```

We need to *parameterise* the method

Improving ReactionTimeMeasurer (1)

```
public void measureReactionTime() {
    this.measureQuestion("John Quay was the Prime Minister");
    this.measureQuestion("6  x 4 = 23");
    this.measureQuestion("Summer is warmer than Winter");
    this.measureQuestion("Wellington's population > 1,000,000 ");
}
public void measureQuestion(String fact) {
    long startTime = System.currentTimeMillis();
    UI.askString("Is it true that" + fact);
    long endTime = System.currentTimeMillis();
    UI.printf("You took %d milliseconds \n",  (endTime - startTime) );
}
```

Understanding ReactionTimeMeasurer

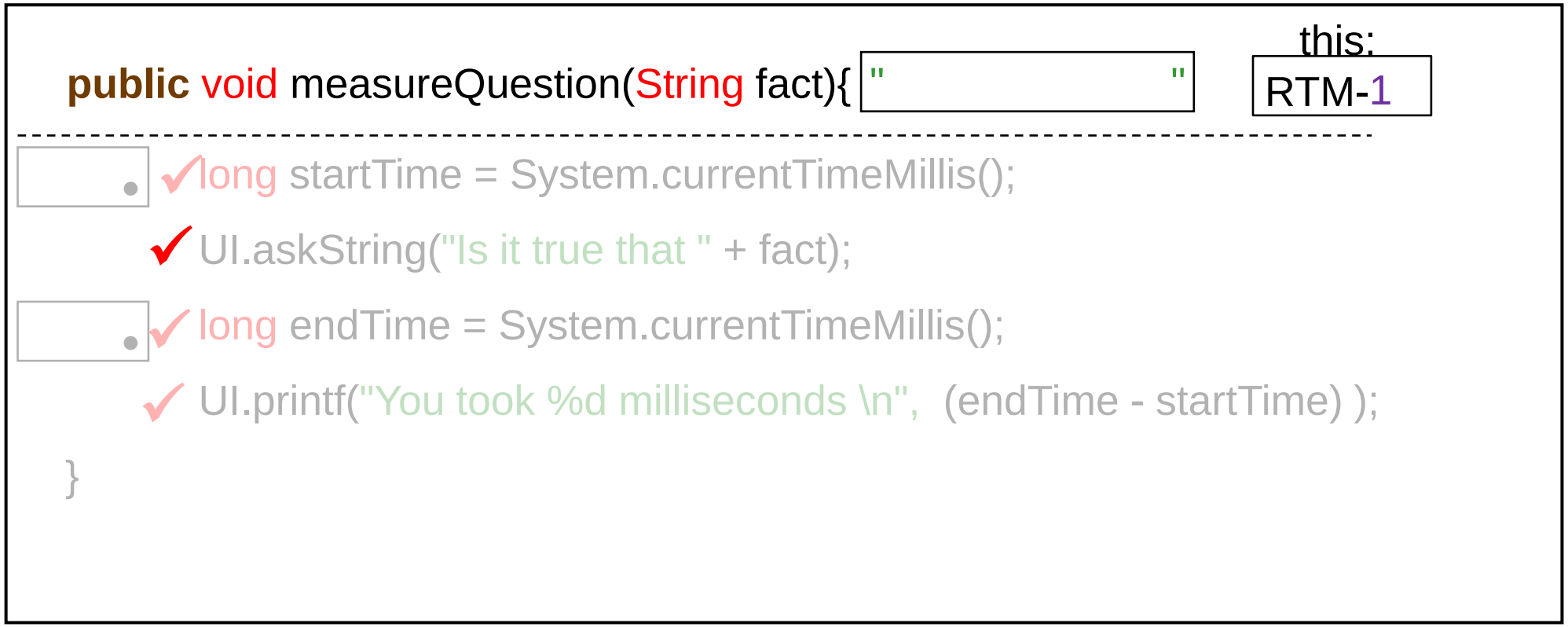
- What happens if we call the method on the object RTM1:

  RTM1 . measureTime();

```
public void measureReactionTime(){                    this:
                                                      RTM- 1
-------------------------------------------------------------------
    this.measureQuestion("John Quay was the Prime Minister");
    this.measureQuestion("6 x 4 = 23");
    this.measureQuestion("summer is warmer than Winter");
    this.measureQuestion("Wellington's population >1,000,000");
```

The object the method was called on is copied to "this" place

Understanding method calls



Understanding ReactionTimeMeasurer

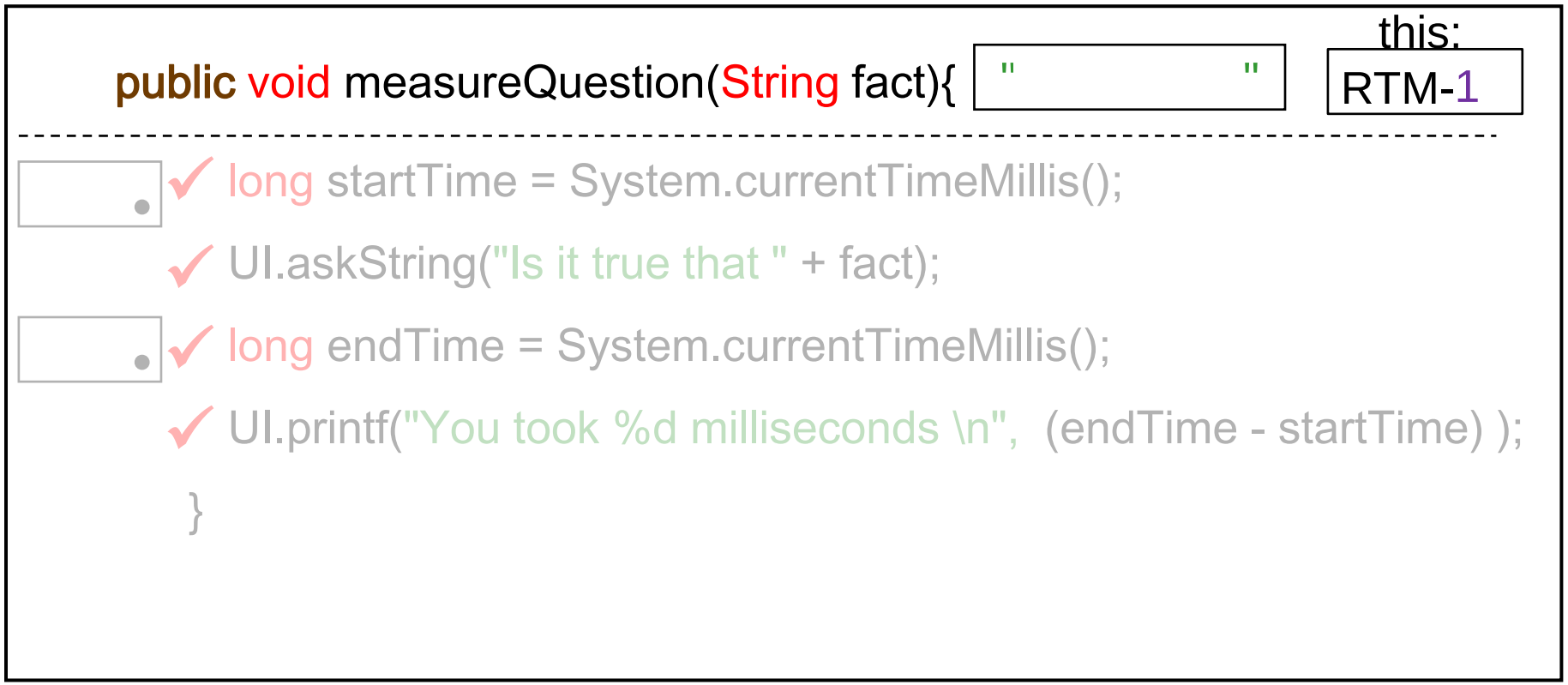
```
public void measureReactionTime(){
```

this: RTM-1

```
✓ this.measureQuestion("John Quay was the Prime Minister");
  this.measureQuestion("6 x 4 = 23");
  this.measureQuestion("summer is warmer than Winter");
  this.measureQuestion("Wellington's population > 1,000,000");
```

Understanding ReactionTimeMeasurer

New measureQuestion worksheet:



Each time you call a method,
it makes a fresh copy of the worksheet!

Understanding ReactionTimeMeasurer

```
public void MeasureReactionTime(){
```

this: RTM-1

✓ this.measureQn("John Quay was the Prime Minister");

✓ this.measureQn("6 x 4 = 23");

this.measureQn("summer is warmer than Winter");

this.measureQn(" Wellington's population > 1,000,000");

Welcome Back!

Restructuring the course for the remaining 9 weeks:

- Lectures all by recorded videos.
  - I intend to record a series of shorter chunks
    - Easier to watch
    - Easier to search for relevant sections for doing the assignments
  - Will be linked from the CourseSchedule page on the course wiki pages.
  - Also accessible via Blackboard

- Still have labs and help desks,
  - via Zoom, https://vuw.zoom.us/my/comp102
  - at regularly scheduled times
  - online help via comp102-help@ecs.vuw.ac.nz

Welcome Back!

Assessment scheme changed:

- No tests or exam
  - We can't run tests or exams in person.
  - Therefore, doesn't make sense to have them at all.
  - Replace the contribution of the tests and exam by raising the weight of the assignments.

- Assignments 3-10 worth 12% each (instead of 2.2%)
  - Same program-style questions for 3-9, as planned, but worth more to your final grade.
  - "Reflection" question → test-style question (1.2% each)
  - Assignment 10 → all test-style questions.

  - Watch the demo videos for all assignments (since it is difficult to access the lab computers!)

ReactionTimeMeasurer Problem

- A good experiment would measure the average time over a series of trials
  - Our program measures and reports for each trial.

- Need to add up all the times, and compute average:
  - problem:
    - MeasureReactionTime needs to add up the times
    - MeasureQuestion actually measures the time, but prints it out.
    - How do we get the time back from MeasureQuestion to MeasureTime?

  - We need to make MeasureQuestion return the time value to MeasureTime.

Methods that return values

- Some methods just have "effects":

```
UI.println("Hello there!");
UI.printf("%4.2f miles is the same as %4.2f km\n", mile, km);
UI.fillRect(100, 100, wd, ht);
UI.sleep(1000);
```

- Some methods just return a value:

```
long  now = System.currentTimeMillis();
double distance = 20 * Math.random();
double ans = Math.pow(3.5, 17.3);
```

- Some methods do both:

```
double height = UI.askDouble("How tall are you");
Color col =JColorChooser.showDialog(UI.getFrame(),  "paintbrush",   Color.red);
```

Defining methods to return values

Improving ReactionTimeMeasurer:

make measureQuestion **return** a value instead of just printing it out.

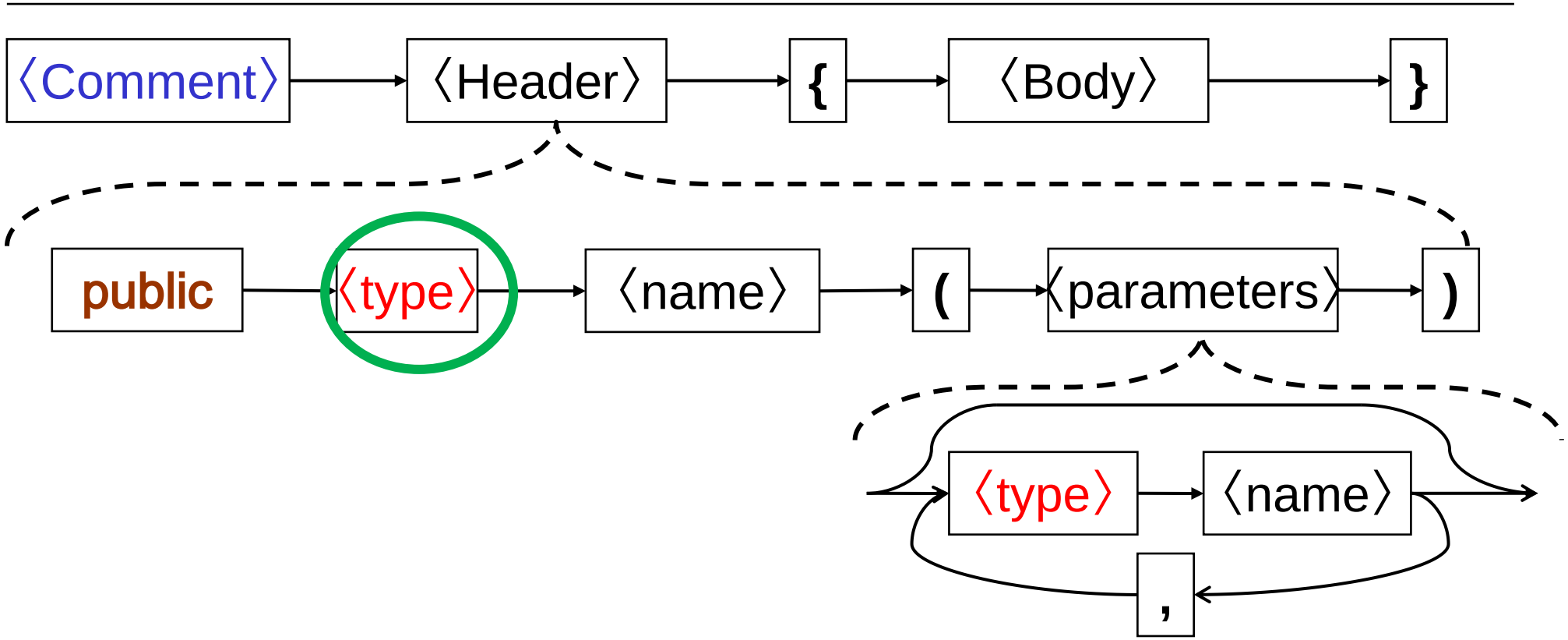
```
public void measureReactionTime() {
    long time = 0;
    time = time + this.measureQuestion("John Quay was the Prime Minister");
    time = time + this.measureQuestion("11 x 13 = 143");
    time = time + this.measureQuestion("Summer is warmer than Winter");
    time = time + this.measureQuestion(" Wellington's pop > 1,000,000 ");
    UI.printf("Average reaction time = %d milliseconds\n", (time / 4));
}
```

Specifies the type of value returned.
void means "no value returned"

```
public long measureQuestion(String fact) {
    long startTime = System.currentTimeMillis();
    ……
}
```

Syntax: Method Definitions (v3: return type)

```
/** Measure time taken to answer a question*/
        public long measureQuestion ( String fact ){
                long startTime = System.currentTimeMillis();
                    :
```



Defining methods to return values

If you declare that a method returns a value,
then the method body must return one!

```
public  long  measureQuestion(String fact) {
    long startTime = System.currentTimeMillis();
    String  ans = UI.askString("Is it true that " + fact);
    long endTime = System.currentTimeMillis();
   return  (endTime - startTime) ;
}
```

New kind of statement
Means: exit the method and return the value
The value must be of the right type

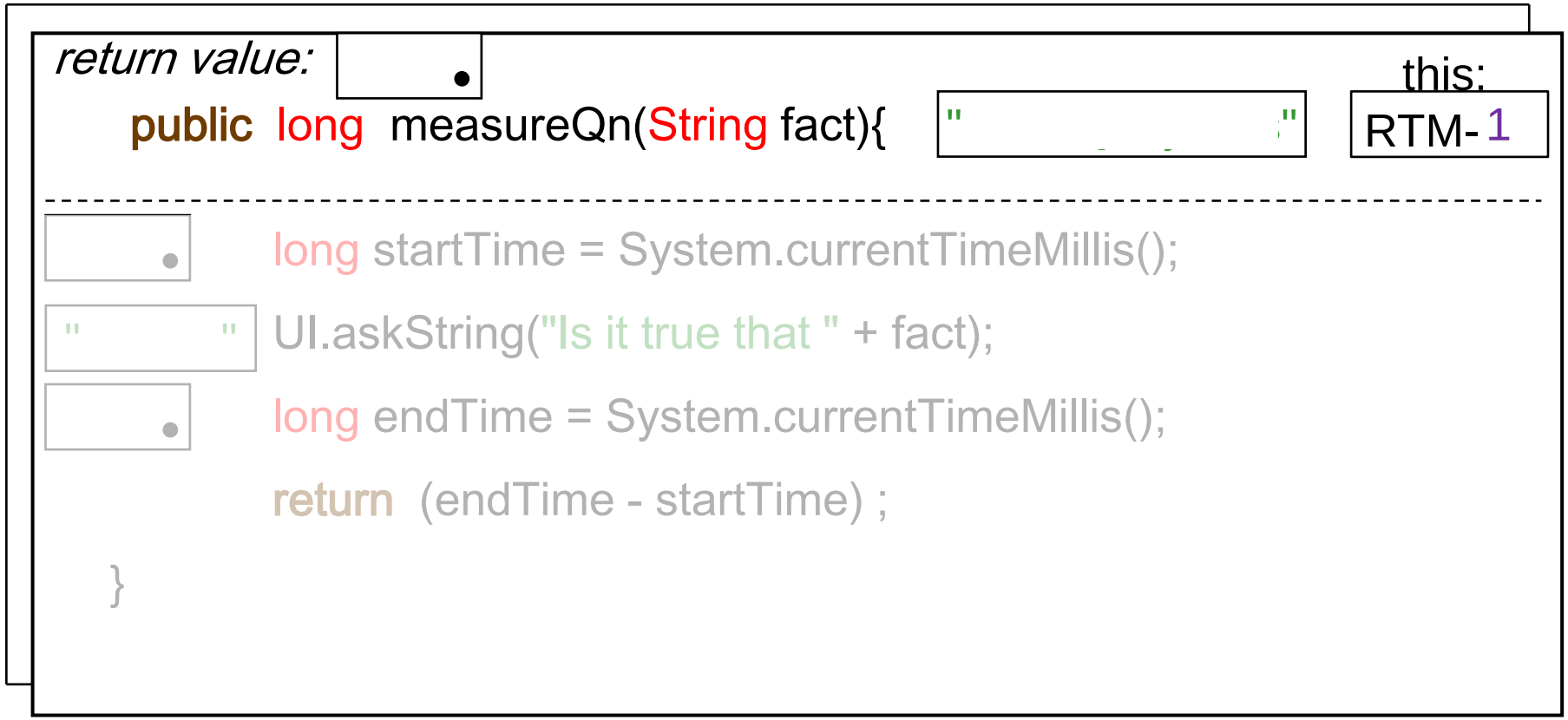
Returning values.

- What happens if we call the method:

  RTM-1 . askQuestions();

```
public void measureReactionTime(){                          this: RTM-1
-------------------------------------------------------------------------
✓ long time = 0;                                    0 •
    time = time + this.measureQuestion("John Quay was the Prime Minister");
    time = time + this.measureQuestion("6 x 4 = 23");
    time = time + this.measureQuestion("summer is warmer than Winter");
    time = time + this.measureQuestion("Wellington's pop > 1,000,000");
```

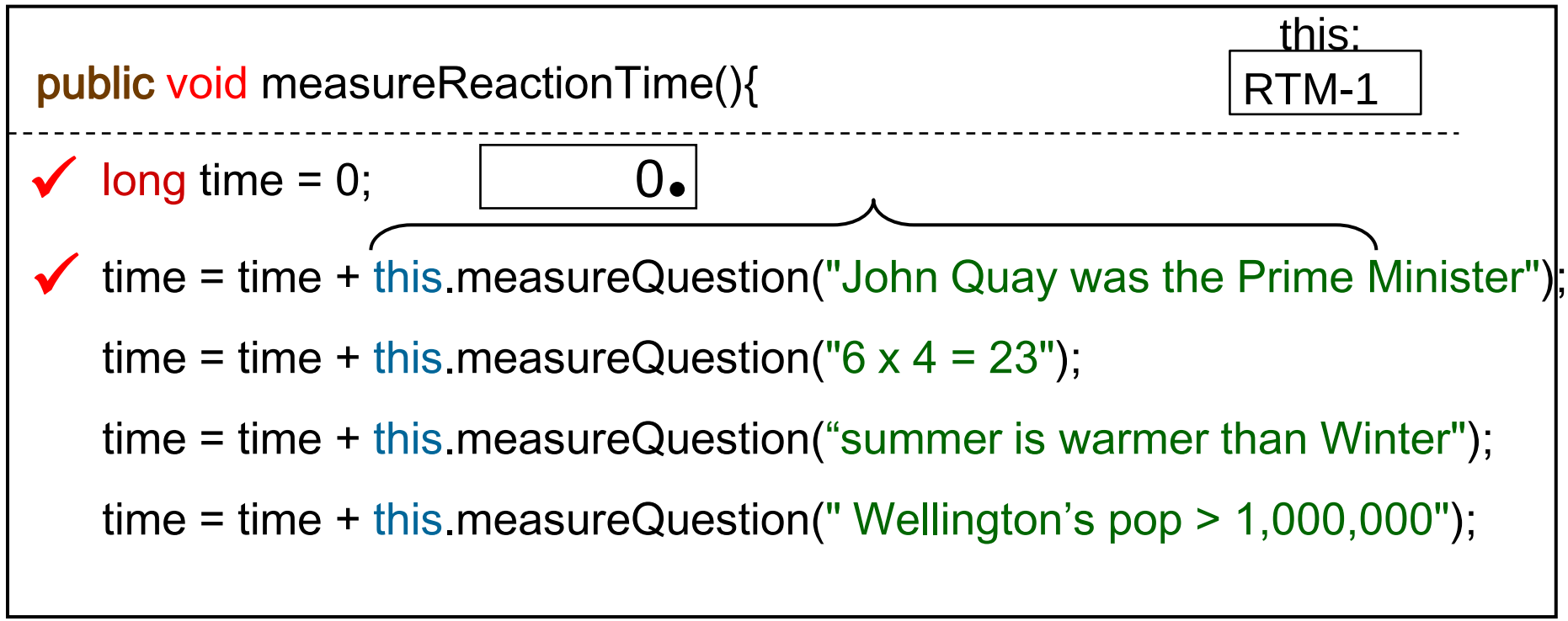
Returning values



Returning values.

- What happens if we call the method:

  RTM-1 . askQuestions();



More about Return

- If a method has a return type, it must have a **return** statement that returns a value

- It must return a value for every possible path

  ⇒ may need several **return** statements:

```
public String fullDayName(String str){
    str = str.toLowerCase();
    if (str.startsWith("m")){
        return "Monday";
    }
    else if (str.startsWith("tu")){
        return "Tuesday";
    }
    else if (str.startsWith("w")){
        return "Wednesday";
    }….
```

More about Return

- **return** does two things:
  - specifies the value that will be returned to the calling method
  - exits the current method, skipping over all remaining statements.

- Methods with a void return type:
  - Can't return a value
  - Can still have a **return** statement (**return**;) with no value.

    ⇒ exit method at this point.

```
public void drawLollipop(double x, double y, double size, double length,){
    if (size < 2 || length < size/2){     // invalid parameters
            return;
    }
    // draw the lollipop
    UI.setColor(Color.red);
    UI.fillRect(x-size/2; y-size/2, size, size);
      :
```

Aside: Random numbers

- Math.random() computes and returns a random double
  - between 0.0 and 1.0

- To get a random number between min and max:
  - min + random number * (max-min)

```
(50.0 + Math.random() * 70.0)
```

gives a value between 50.0 and 120.0

- This is an expression:
  - can assign it to a variable to remember it
  - can use it inside a larger expression
  - can pass it directly to a method

Menu

- Repetition/Iteration
  - doing something to each value in a list ("for each" loop)
  - doing something to a sequence of numbers ("for" loop)
  - repeating something as long as a condition stays true ("while" loop)

Repetition / Iteration

Doing some action repeatedly:

- “Polish each of the cups on the shelf”
- “Put every chair on top of its desk”
- “Give a ticket to everyone who passes you”
- “Keep running around the track until 6pm”
- “Practice the music until you can play it perfectly”

Two patterns:

- Do something to each thing in a collection
- Do something until some condition changes

Repetion/Iteration in Java LDC 4.5

Several different ways of specifying repetition.

- **For** Each statement: Do something to each element of a list

```
for ( type variable : listOfValues ) {
    do something to value in variable
}
```

- Counted **For** statement: Do something to each number from …..

```
for ( int num = ⟨start⟩;  num <= ⟨end⟩ ;  num = num + ⟨increment⟩ ) {
    do something with num
}
```

- **While** statement: Repeat some action while some condition is still true

```
while (condition-to-do-it-again ) {
    actions to perform each time round
}
```

For Each statement

Three components

- a list of values
- a variable that is assigned each value of the list in turn.
- actions to perform for each value in the list

```
// print each number in a list of numbers:
for ( double num : listOfNumbers ) {
    UI.println(num);
}
```

listOfNumbers: 150.0, 32.2, 6.9, 49.5, 83.4, -21.0, 1.0

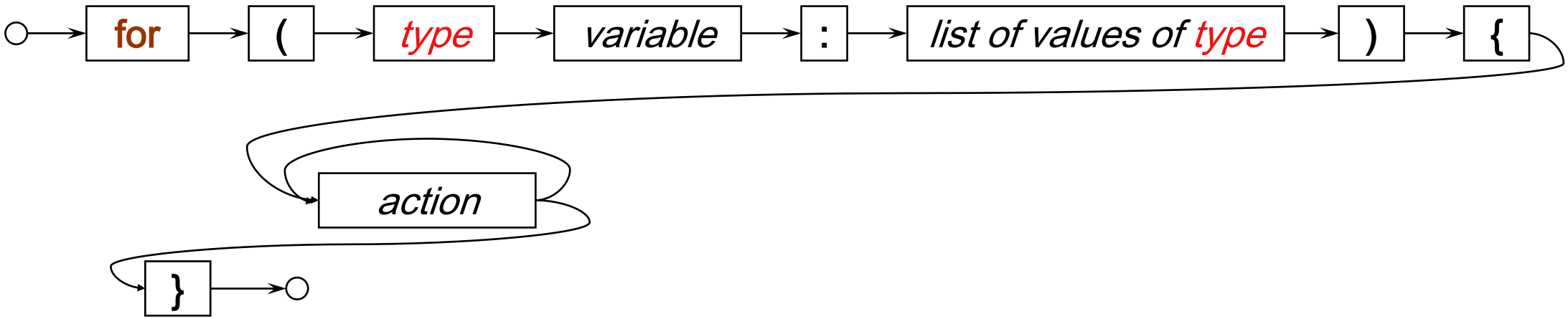
num:

```
// print each string in a list of numbers that starts with "A":
for ( String str : listOfStrings ) {
    if ( str.startsWith("A") ) {
        UI.println(str);
}
```

listOfStrings: "Jamie", "Andie", "Jules", "Amy", "Mark"

str: " "

For statement ("for each" version)




```
for ( double num : listOfNumbers ) {
    UI.println(num);
}
```

- Meaning:

  Repeatedly (for each value in the list)
  - put the next value of the list into the variable
  - do the actions.

Lists of values

- What type is a **list** of values?
- How do we get a list of values?

List of doubles

Have to use Double, not double
Double is the "wrapped-up" version of double, for putting into a list

```
ArrayList <Double>  numberList   =  UI.askNumbers("Enter numbers");
for (double num : numberList)  {
    UI.println(num);
}
UI.setColor(Color.red);
UI.setLineWidth(5);
for (double radius : numberList)  {
    if (radius> 20  &&  radius < 200) {
        UI.drawOval( 300 – radius,  250 – radius,  radius * 2.0, radius * 2.0);
}
```

Asks for a list of numbers, ending with 'done'

Lists of values

- What type is a **list** of values?
- How do we get a list of values?

List of String values

```
ArrayList <String>  nameList   =  UI.askStrings("Enter names");
for (String name : nameList)  {
    UI.println("Hello " + name);
}
UI.println("============ Long names ============");
for (String name : nameList)  {
    if (name.length() > 6 ) {   UI.println(name);    }
}
UI.println("============ Short names ============");
for (String name : nameList)  {
    if (name.length() <= 6 ) {   UI.print(name + ", ");    }
}
UI.println();
```

Asks for a list of strings, ending with empty line

print without a new line

print just a new line

Doing more with the loops: using Variables

- Add up all the numbers in a list:

numberList: 150.0, 32.2, 6.9, 49.5, 83.4, -21.0, 1.0

```
ArrayList <Double>  numberList   =  UI.askNumbers("Enter numbers");


double total = 0.0;
for (double num : numberList)  {
    total = total + num;
}
UI.println("Total of numbers = " + total );
```

Declare and initialise variable

total:

num:

Add each number into the total:
- Uses current value in total
- Adds the next number to it
- Puts result back into total

Doing more with the loops: using Variables

- Count the number of long names in a list.

```
ArrayList <String>  nameList   =  UI.askStrings("Enter names");

int count = 0;

for (String name : nameList)  {
    if (name.length() > 6 ) {
        count = count + 1;
    }
}
UI.printf("There were %d long names out of %d names \n", count,  nameList.size() );
```

Declare and initialise variable

Add 1 to the count

Number of values in a list

Lists are values too: passing lists around

```
public void analyseNames() {
    ArrayList <String>  nameList   =  UI.askStrings("Enter names");
    UI.println("Total characters: "  + this.totalChars (nameList) );
    UI.println("Starts with A: " + this.wordStartingWith(nameList, "A") );
}
public int totalChars(ArrayList <String>  strings ){
    int count = 0;
    for (String str : strings)  {
        count = count + str.length();
    }
    return count;
}
public String wordStartingWith(ArrayList <String>  strings,  String pattern ){
    for (String str : strings)  {
        if ( str.startsWith(pattern) )  {  return str;  }      // returns first word starting with the pattern
    }
    return "<none>";
}
```

Counted For statement

Alternative form of the for statement.

More general, more powerful, more tricky to get right

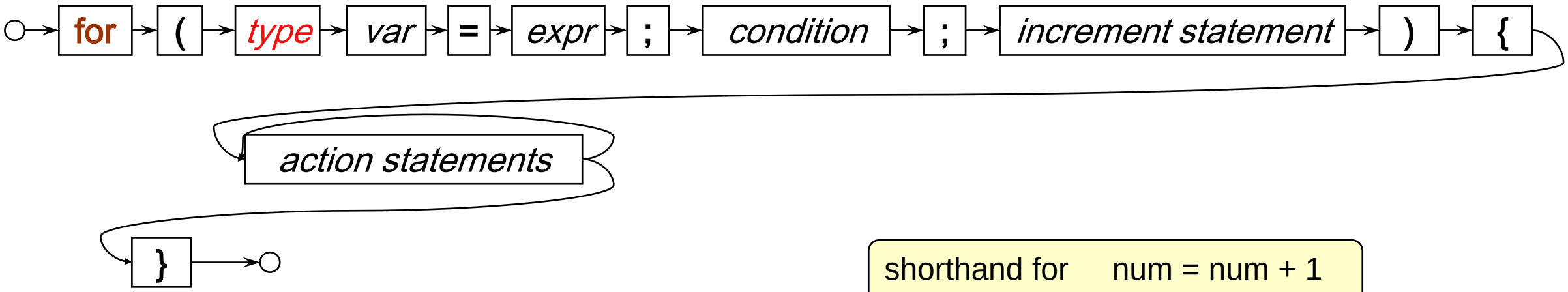
Four components

- a variable and its initial value.
- a condition when to keep going / stop
- how to increment the variable each time
- actions to perform for each time

num: 

```
// print each number from 1 to 100:
for ( int num =1;   num <= 100;   num = num + 1 )  {
      UI.println(num);
}
```

For statement ("counted" version)



shorthand for num = num + 1

```
for ( int num = 0 ; num < 1000 ; num++ ) {
    UI.println(num);
}
```

- Meaning:
  - initialise the variable
  - repeat, as long as the condition is true:
    - do the actions
    - do the increment

Using Counted For: #1

- Print a table of numbers and their squares:

```
public void printTable(int max){
    UI.println("Table of integers and their squares");
    for (int num = 1;   num <= max;   num = num + 1 ) {
       UI.printf(" %3d    %6d   %n",  num,  (num*num));
    }
}
```

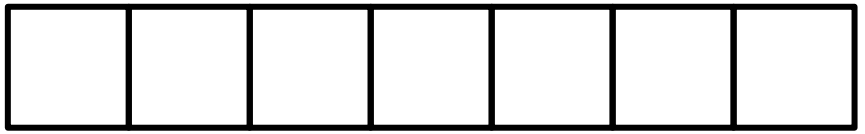
Using Counted For: #2

Doesn't have to increment by 1 each time:

```
/**
 * Print each even number between start and end (inclusive)
 */
public void printEvenNumbers(int start, int end ){
    if (start%2==1 ) {    // make sure start is even
        start = start + 1;
    }
    for ( int num = start;   num <= end;  num = num + 2 )  {
        UI.println(num);
    }
}
```

Using Counted For: #3

- Draw a row of squares:



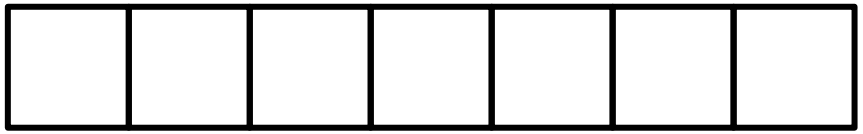
```
public static final double SIZE = 20;
   ⋮

/** Draws count squares in a horizontal row, starting at (left,top)  */
public void drawRowOfSquares (double left, double top, int count){
    for  (int i = 0;  i < count;   i++  ) {
        double x = left + i * SIZE;
        UI.drawRect(x, top, SIZE, SIZE);
    }
}
```

i++
is shorthand for
i = i + 1

Counting from 0 is often easier, especially for drawing stuff!

Using Counted For: #4

- Draw a row of squares:



```
public static final double SIZE = 20;
   ⋮

/** Draws count squares in a horizontal row, starting at (left,top)  */
public void drawRowOfSquares (double left, double top, int count){
    double right = left+count*SIZE;
    for  (double x = left;   x < right;   x = x + SIZE) {
        UI.drawRect(x, top, SIZE, SIZE);
    }
}
```

Not really a "Counted for"!

Using Counted For: #5

- For doesn't have to step up:

```
public void countDown(int start){
    UI.setFontSize(100);
    for (int count = start;  count >= 1;  count = count – 1) {
        UI.clearGraphics();
        UI.drawString( ""+count,  200, 300 );
        UI.sleep(500);
    }
    UI.clearGraphics();
    UI.setColor(Color.red);
    UI.drawString("GO",  200, 300);
}
```

could use shorthand:
count - -

Count from 0 or 1?

Counted for loop: Can count from 0 or from 1

```
for (int n = 0; n < target; n++) {
    ⟨do actions⟩
}
```

OR

```
for (int n = 1; n <=max;  n++) {
    ⟨do actions⟩
}
```

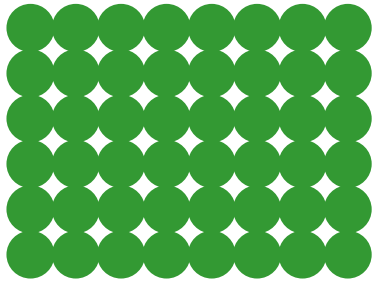
- If counting from 0,
  - n is the number of iterations that have been completed
  - Loop as long as n is less than target:
  - Good for drawing
  - Good for dealing with lists and arrays.
- If counting from 1,
  - n is the iteration it is about to do
  - Loop as long as n is less than or equal to target:

Off-by-one errors are common when you mix these two up.

Nested for loops

Can have loops inside loops:

eg: Draw a grid of circles



```
public void drawCircles(int  rows,  int cols, int diam ) {
    for (int row = 0;  row < rows; row++) {
        double y = TOP +row*diam;
        for (int col = 0;  col < cols; col++) {
            double x = LEFT + col*diam;
            UI.fillOval(x, y, diam, diam);
        }
    }
}
```

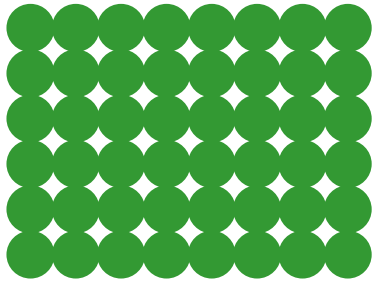
Outside loop:
do each row

Inside loop:
do each column within the current row

Nested for loops

Nested loops can be row first, or column first:

eg Draw a grid of circles (by column)



```
public void drawCircles(int  rows,  int cols, int diam ) {
    for (int col = 0; col < cols; col++) {
        double x = LEFT + col*diam;
        for (int row = 0; row < rows; row++) {
            double y = TOP +row*diam;
            UI.fillOval(x, y, diam, diam);
        }
    }
}
```

Outside loop:
do each column

Inside loop:
do each row within the current column

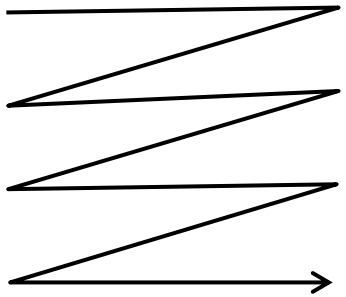
Designing nested loops with numbers

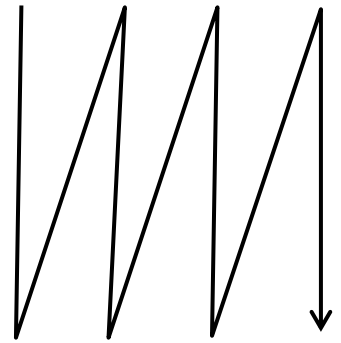
2D structures, eg table of rows and columns:

- Can do rows in the outside loop and columns in the inside loop, or vice versa

```
for (int row=0;row<rows; row++) {
    for (int col=0; col<cols; col++ ) {
        ⟨do actions for row, col ⟩
    }
}
```

```
for (int col=0; col<cols; col++) {
    for (int row=0;row<rows row++) {
        ⟨do actions for row, col ⟩
    }
}
```





While statements: repeating with a condition

- **For** statements: repetition over a list of values.
- **While** statements : general repetition, subject to a condition.

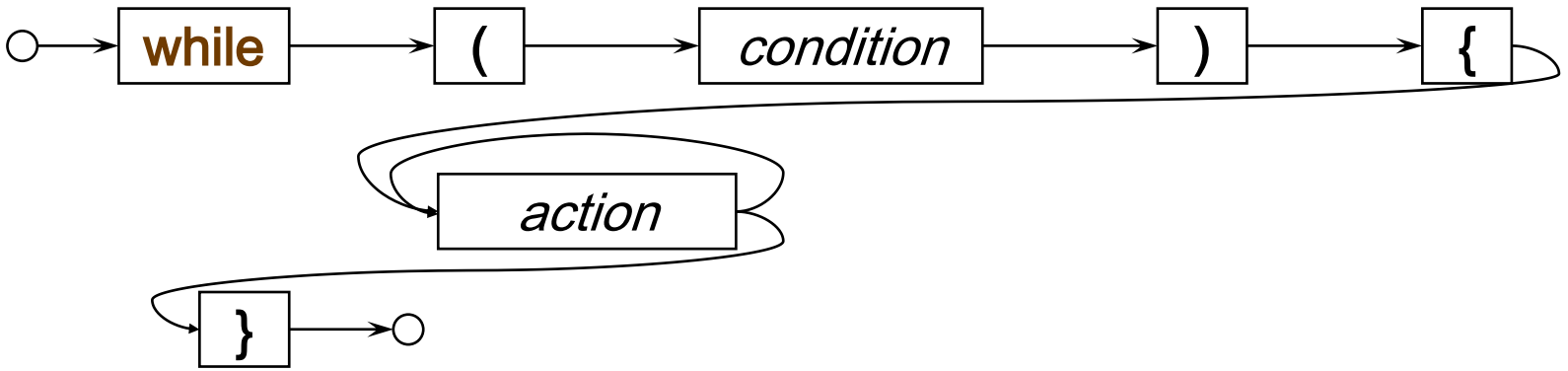
```
while (condition-to-do-it-again ) {
    actions to perform each time round
  }
```

Similar structure to the **if** statement

```
while ( true ) {
    UI.println("this repeats forever!");
}
```

- Similar to **for**, but NOT THE SAME!
  - same condition and actions;
  - no built-in initialisation and increment.
  - Appropriate if you don't know how many times it will repeat

While statement



- Meaning:

  Repeatedly
  - If the condition is still true, do the actions another time
  - If the condition is false, stop and go on to the next statement.
    - Note: don't do actions at all if the condition is initially false

- Similar to **if**, but NOT THE SAME!
  - keeps repeating the actions,
    - as long as the condition is still true each time round
  - no **else** — just skips to next statement when condition is false

While is a rearrangement of for

- Print a table of numbers and their squares:

```
public void printTable(int max){
    int num = 1;                          Initialise
    while ( num <= max ) {                Test
        UI.printf(" %3d    %6d   %n",  num,  (num*num));   Body
        num = num + 1;                    Increment
    }
}
```

- Repetition with **while** generally involves
  - initialisation: get ready for the loop — Put before the while loop
  - test: whether to repeat
  - body: what to repeat
  - "increment": get ready for the next iteration — Put at the end of the actions.

General while loops

```
/** Practice times-tables until got 5 answers correct */
public void playArithmeticGame (){
    int score = 0;
    while ( score  < 5) {
        // ask an arithmetic question
        int a = this.randomInteger(10);
        int b = this.randomInteger(10);
        int ans = UI.askInteger("What is " + a + " times " + b + "?" );
        if ( ans ==  a * b ) {
            score = score +1;
        }
    }
    UI.println("You got 5 right answers" );
}
```

```
public int randomInteger(int max) {
    return  (int) (Math.random() * max ) + 1;
}
```

General while loops

```
/** Ask a multiplication problem until got it right */
public void practiceArithmetic (){
    int a = this.randomInteger(10);
    int b = this.randomInteger(10);
    String question = "What is " + a + " times " + b + "?";
    boolean correct = false;
    while ( ! correct) {
        int ans = UI.askInteger(question);
        if ( ans ==  a * b ) {
            correct = true;
            UI.println("You got it right!" );
        }
        else {
            UI.println("sorry, try again");
        }
    }
}
```

This seems unnecessarily complex!!

Loops with the test "in the middle"

If the condition for exiting the loop depends on the actions, need to exit in the middle!

Common with loops asking for user input.

- **break** allows you to exit a loop (**while**, or **for**)
  - Must be inside a loop
  - Ignores any **if** 's
  - Does not exit the method (**return** does that)

```
while ( true ) {
    actions to set up for the test
    if ( exit-test ) {
        break;
    }
    additional actions
}
```

General while loops with break

```
/** Ask a multiplication problem until got it right */
public void practiceArithmetic (){
    int a = this.randomInteger(10);
    int b = this.randomInteger(10);
    String question = "What is " + a + " times " + b + "?";

    while ( true ) {
        int ans = UI.askInteger(question);
        if ( ans ==  a * b ) {
            UI.println("You got it right!" );
            break;
        }
        UI.println("sorry, try again");
    }
}
```

Setting up for test

Test and break

actions if test failed.

- Only use break when the exit is not at the beginning of the loop.

While loops to get valid input

```
/** Ask for an email address and insist that it contains one @ in the middle */
public String askEmailAddress (){
    String addr = UI.askString("Enter email address");
    int posAt = addr.indexOf("@");
    while ( posAt <= 0
            ||  posAt == addr.length()-1
            ||  addr.indexOf("@", posAt+1) > -1
            ||  addr.contains(" ") ) {
        UI.println("Email address must have a single @ with characters before and afterwards");
        addr = UI.askString("Enter email address");
        posAt = addr.indexOf("@");
    }
    return addr;
}
```

Duplicate to do action before the first test

While loops to get valid input

```
/** Ask for an email address and insist that it contains one @ in the middle */
public String askEmailAddress (){
    while (true) {
        String addr = UI.askString("Enter email address");
        int posAt = addr.indexOf("@");
        if ( posAt > 0  &&   posAt < addr.length()-1
                &&  ! addr.contains(" ")
                &&  addr.indexOf("@", posAt+1) == -1 ) {
            return passwd;
        }
        UI.println("Email address must have a single @ with chara erwards");
    }
}
```

Actions before test

Actions after test

While loops to get valid input

```
/** Ask for an email address and insist that it contains one @ in the middle */
public String askEmailAddress (){
    while (true) {
        String addr = UI.askString("Enter email address");
        if ( addr.contains("@")  &&  ! addr.startsWith("@") &&  ! addr.endsWith("@")
              &&  ! addr.contains(" ")
              &&  addr.indexOf("@", addr.indexOf("@")+1) == -1  ) {
            return passwd;
        }
        UI.println("Email address must have a single @ with characters before and afterwards");
    }
}
```

Actions before test

Actions after test

More loops with user input

- Make user guess a magic word:

```
public void playGuessingGame(String magicWord){
    UI.println("Guess the magic word:");
    while (true) {
        String guess = UI.askString("your guess: ");
        if  ( guess.equalsIgnoreCase(magicWord) ) {
            UI.println("You guessed it!");
            break;
        }
        UI.println("No, that wasn't right. Try again!");
    }
}
```

Setting up for test

Test and break

Additional actions

Testing your program

A) Need to try out your program on sample input while removing the "easy" bugs.

- Can be a pain if need lots of input (eg TemperatureAnalyser)
- UI window has a menu item – "set input" – to get input from a text file instead of user typing it.

  ⇒ don't have to type lots of data each time
  - Create the text file, eg in Notepad
  - Select file using menu *before* the program has started asking for input.
  - File can contain multiple sequences of data.

- B) Need to test your program on a range of inputs
  - Easy, "ordinary", inputs
  - Boundary cases — values that are only just in range, or just out of range
    Need to check that your **if** conditions are right
  - Invalid data—does your program handle invalid input correctly?

  Creating test cases involves creativity – have to try to come up with ways to break your program.