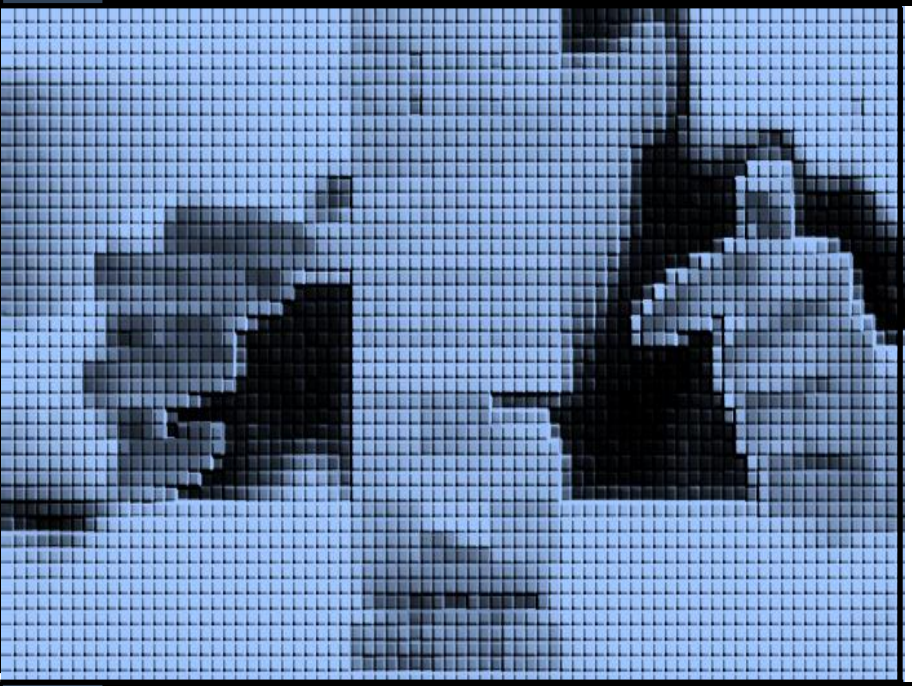
University of BRISTOL

School of Computer Science

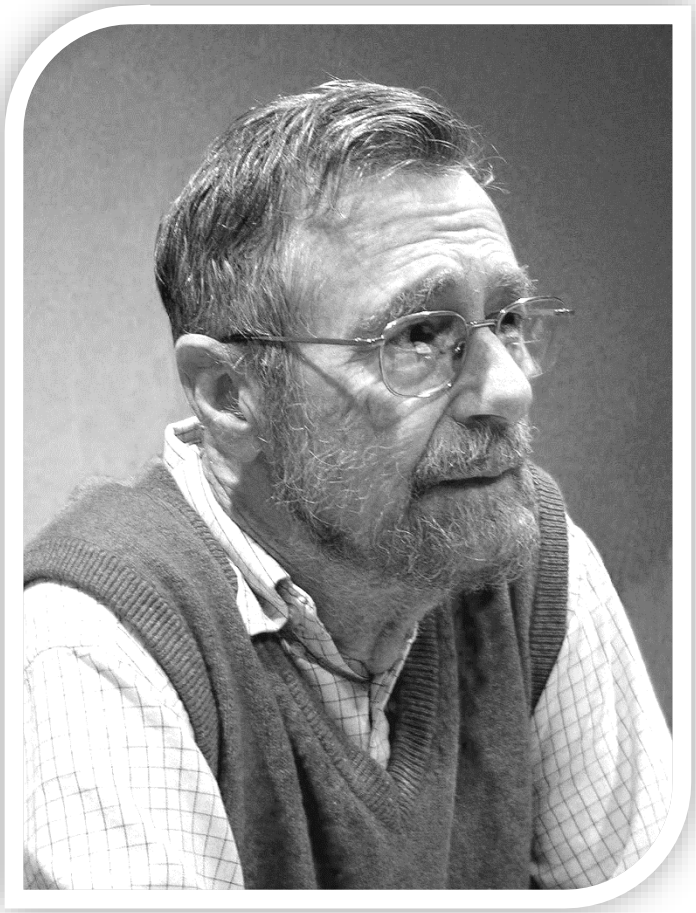
Programming in Java



Java Programming – Lecture 04

Loops & Jumps

Sion Hannuna and Simon Lock,
based on Tilo Burghardt's C Unit

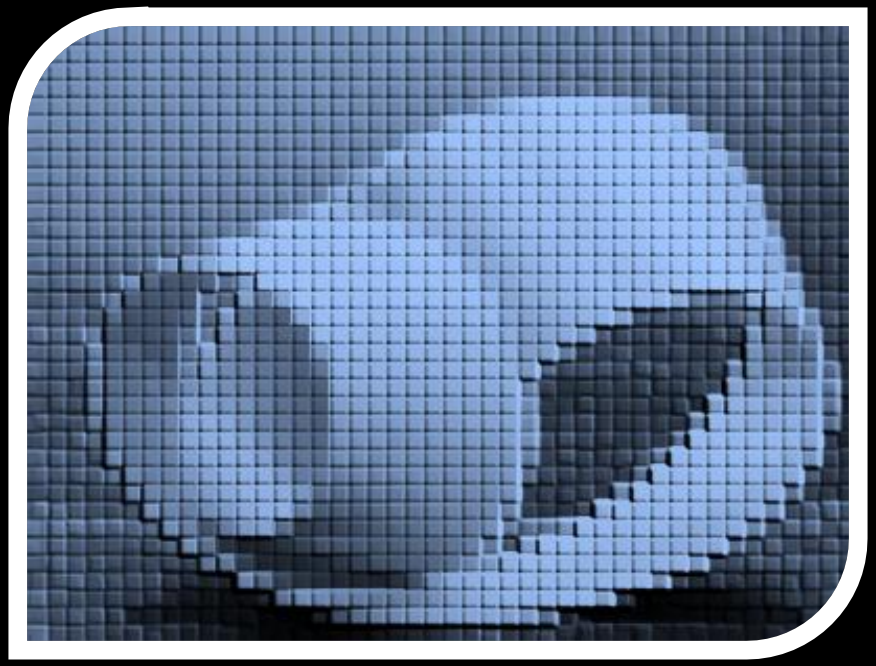


(c) H Richards

“ The art of programming is the art of organizing complexity. ”

Edsger W Dijkstra (1930-2002)
computer scientist

While-Loops



Countdown

```
/* Print a countdown in 1s intervals. */
public class Countdown {
   public static void main(String[] args) throws java.lang.InterruptedException {
      int t = 10;
      while (t > 0) {
         Thread.sleep(1000); // wait 1 second
         System.out.println(t);
         t--;
      }
   }
}
```

Countdown.java

- A *while loop* allows statements in a following block to be repeated:

```
while (EXPRESSION) {
   STATEMENTS;
}
```

- As long as the expression is *true* the block is executed again and again…
- `while` combines a condition and a backward jump from the block's end to its start.

Increment and Decrement Operators

- There are increment and decrement abbreviations:

```
n++;          // means n = n + 1;
++n;          // means n = n + 1;
n += 1;       // means n = n + 1;
n--;          // means n = n - 1;
--n;          // means n = n - 1;
n -= 1;       // means n = n - 1;
m = n++;      // means m = n; n++;
m = ++n;      // means n++; m = n;
n = n++;      // undefined, this is a bug
n = ++n;      // undefined, this is a bug
n = n--;      // undefined, this is a bug
n = --n;      // undefined, this is a bug
```

```
...
  int t = 10;
  while (t > 0) {
    sleep(1);
    printf("%d\n", t);
    t--;
  }
  return 0;
...
```

Another Example: Square Root

- For a given floating point number x, calculate its square root via Newton's Algorithm:

```
/* Find the square root. */
import java.util.Scanner;

public class SquareRoot {
   // Square root (like sqrt) via Newton Algorithm
   public static double root(double x) {
      double r = x / 2.0; // first guess
      double epsilon = 1E-14;
      while (Math.abs(r - x / r) > epsilon) {
         r = (r + x / r) / 2.0; // Newton step
      }
      return r;
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter number:");
      double x = scanner.nextDouble();
      System.out.printf("The square root is: %f\n", root(x));
   }
}
```

SquareRoot.java

Understanding the Algorithm

- After a first guess `r`, we repeat Newton steps `while` the gap between our guess and `x/r` is bigger than some small `epsilon` (it must be `0` for the square root itself).

```
...
public static double root(double x) {
  double r = x / 2.0; // first guess
  double epsilon = 1E-14;
  while (fabs(r - x/r) > epsilon) {
    r = (r + x/r) / 2.0; // Newton step
  };
  return r;
} ...
```

- If `r` is less than the real root, then `x/r` is greater, and vice versa, so the *average* is a better approximation. This essentially defines a Newton step.
- This procedure is easy to understand, convergence is rapid (faster than halving the gap), but libraries use even faster special-purpose techniques (e.g. reciproot).

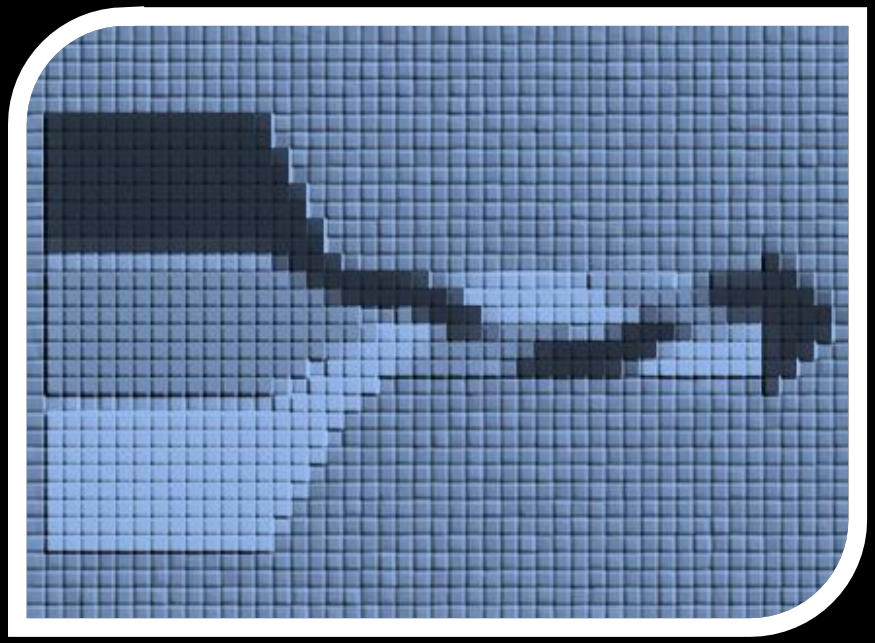
Edge Cases

- When implementing an algorithmic procedure it is particularly important to check the edge cases.

```
...
public static double root(double x){
  double r = x / 2.0; // first guess
  double epsilon = 1E-14;
  while (fabs(r - x/r) > epsilon) {
    r = (r + x/r) / 2.0; // Newton step
  };
  return r;
} ...
```

- In our `root` procedure, an example edge case is when `x = 4.0`, since the initial guess `r` is *exactly* correct.
- All is in order here, since the expression in the `while` loop will turn to be false immediately, and the body of the `while` loop is repeated `0` times.
- The code thus *falls-through* and correctly `return`s `r`.

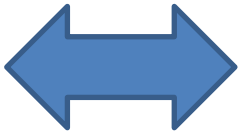
For-Loops



For Loops

- Our countdown loop could be rewritten as a `for` loop:

```
{
  int t = 10;
  while (t > 0) {
    ...
    t--;
  }
} // t out-of-scope
```



```
for(int t = 10; t > 0; t--) {
    ...
} // t out-of-scope
```

- The variable declaration+initialisation `int t = 10;` as well as the continuation condition `t > 0`, and the per-loop variable decrement `t--` are all gathered in *one single line of code* now.

Stylised For Loops

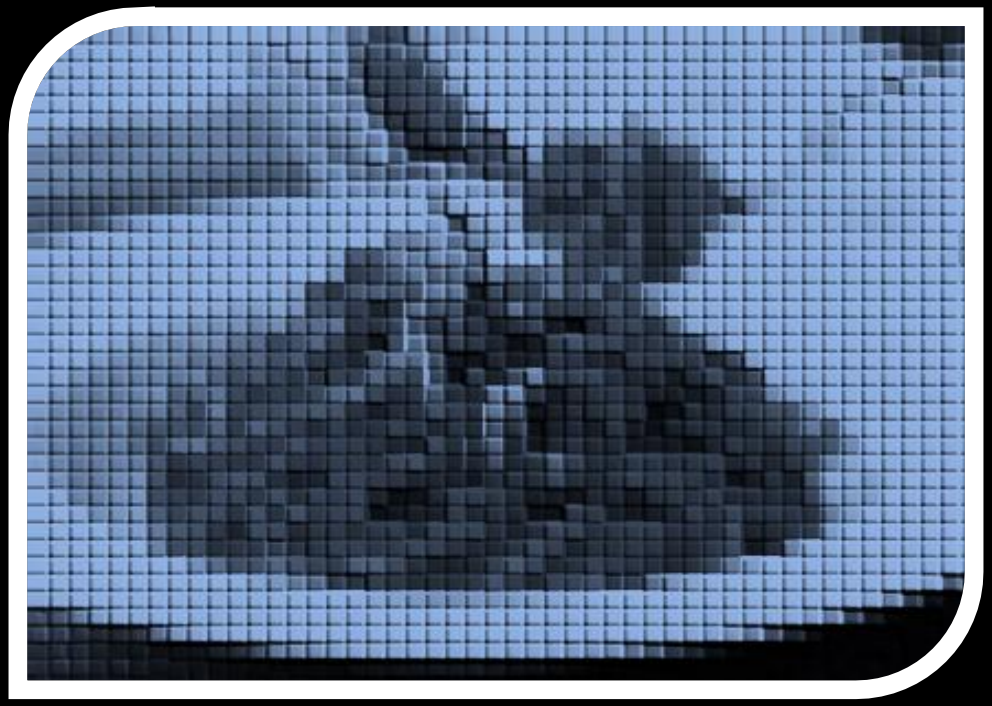
- Because of their logical complexity, you should only use `for` loops in a few familiar stylised special cases, e.g.:

```
for(int i = 0; i < n; i++) { ... }      // count up n times

for(int i = n - 1; i >= 0; i--) { ... } // count down n times
```

- If your situation isn't a simple one like these, it is probably better to use a `while` loop.
- `while` loops *explicitly separate* the three iteration elements and are thus often more readable…

Jumping & Spaghetti Code



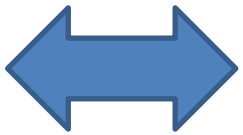
Jumping

- Usually your code executes line-by-line downwards.
- We have seen that: during a function call execution *jumps* to the function and then `return`s in a controlled way.
- We have seen that: standard `while` and `for` loops have *one single* well defined point (at the end of the block) where the code potentially *jumps* back to the block's beginning.
- **Warning:** The more your code jumps about, the harder it is to debug. Code that jumps excessively is *spaghetti code*.
- Thus, whenever possible: control jumps by using function calls, `if` statements, and standard `while` and `for` loops *instead* of the functionality discussed (for completeness only) in the following slides…

Do-While Loops

- Whenever a `while` loop's body is to be executed at least once, then the condition whether to jump back can be tested at the end of the loop.
- A `do-while` loop implements this behaviour:

```
{
  int t = 10;
  do {
    ...
    t--;
  } while (t > 0);
}
```



```
{
  int t = 10;
  while (t > 0) {
    ...
    t--;
  }
}
```

more readable alternative

- Thus, in the above examples, the two loops are only equivalent if `t` is initialised with a value above `0` (like `10`).

Early Loop Restart

- The `continue` statement restarts a loop early:

```
...
// process prime numbers only
for(int i = 0; i < n; i++) {
  if (!isPrime(i)) continue;
  ...
} ...
```

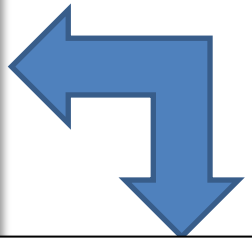
```
...
// process prime numbers only
for(int i = 0; i < n; i++) {
  if (isPrime(i)) {
    ...
  }
} ...
```

more readable alternative

- Using `continue` introduces an extra jumping point in the code and makes control flow more difficult to trace.
- Some programmers would therefore argue that using an `if` statement inside a loop is always better than using `continue`.

Early Loop Exit

```
// Search for first prime in a range
while (i < last) {
  if (isPrime(i)) break;
  i++;
}
```



```
// Search for first prime in a range
while ((i < last) && (!isPrime(i))) {
  i++;
}
```

more readable alternative

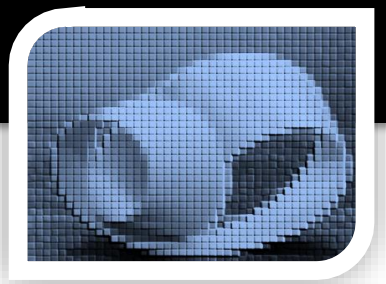
- A break statement exits a loop.
- It introduces an extra jumping point in the code.
- One disadvantage is that the loop can end while the test expression is still true.
- Ideally, a test expression should be what must be true each time round the loop, and false when it ends (making it easier to prove correctness).

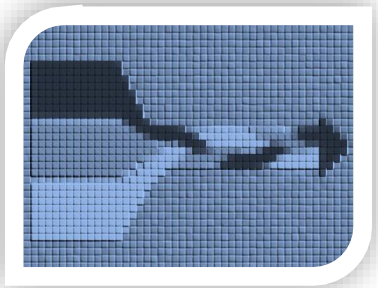
Early Function Return

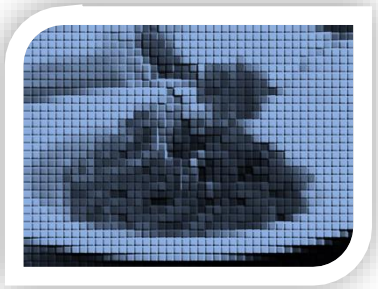
```
...
public static double fabs(double x) {
  if (x >= 0) return x; // returning early here
  return -x;
} ...
```

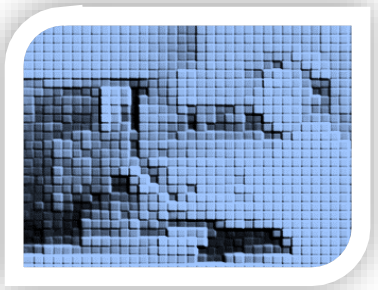
- The `return` statement doesn't have to be at the end.
- No `else` is needed in the code above: `if (n >= 0)`, then execution `return`s from the function before reaching the second line.
- One stylised use is to dispose of an exceptional case.
- The disadvantage is it may be unclear what property holds on return, or how to add extra end-code.

Lecture 04 Summary









IN THIS LECTURE WE COVERED:

- *while loops, edge cases, for loops, do-while loops, break, continue, jumping, spaghetti code, early return*

AFTER THIS LECTURE:

- In your own time: recap the concepts of this lecture and compile, run and comprehend all its programs.

NEXT LECTURE: Arrays