- Variable declaration

  ```
  type variable;
  ```

- Assignment

  ```
  variable = expression;
  ```

- Method invocation

  ```
  objectreference.methodname(parameters);
  ```

  or

  ```
  classname.methodname(parameters);
  ```

- Conditional

  ```
  if (condition) block;
  ```

  or

  ```
  if (condition) block1; else block2;
  ```

- Loop

  ```
  while (condition) block;
  ```

Loops

- The basic loop statement:

```
while (boolean_expression) {
  list_of_statements;
}
```

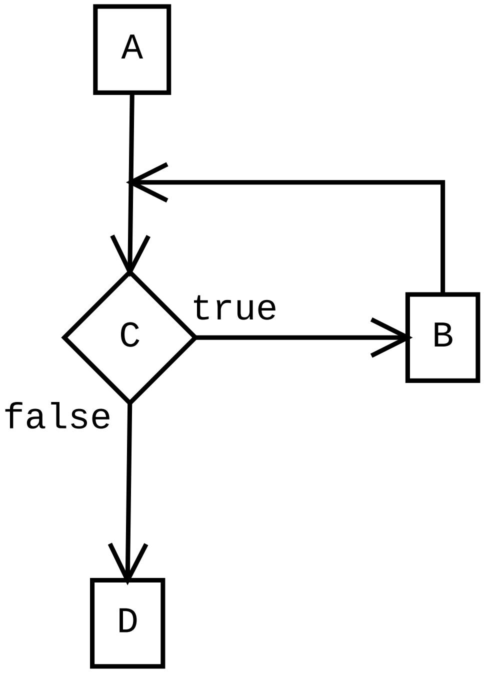
- Semantics: the execution of a while loop proceeds as follows:

1. The boolean expression is evaluated
   (a) If it is false,
       i. the loop stops
       ii. and computation proceeds directly after the loop
   (b) If it is true,
       i. the list of statements is executed,
       ii. and when finished, the whole process is repeated from step 1

Loops

```
A;
while (C) {
  B;
}
D
```

- Control flow diagram:



Loops

```
// This prints the first 100 odd numbers, and
// *not* the odd numbers less than 100.
int counter, number;
counter = 1;
number = 1;
while (counter <= 100) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
System.out.println(“Done”);
```

Loops

```
int maximum = Keyboard.readInt();
int counter = 1;
int number = 1;
while (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
```

Loops

- `while` is *not* the same as `if`

```
int maximum = Keyboard.readInt();
int counter = 1;
int number = 1;
if (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
    counter++;
}
```

- The while statement executes a statement or list of statements *repeteadely*, until its condition becomes false

- The if statement executes a statement or list of statements *once*, and only if its condition is true

Loops

- A loop may not terminate

```
int maximum = Keyboard.readInt();
int counter = 1;
int number = 1;
while (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
}
```

- A loop will not terminate if its condition is always true
- The condition of a loop will remain true if its variables never change

Loops

- The variables of the condition must change in a way which eventually makes the condition false

- If the variables change, but in a way that does not make the condition false eventually, then the loop does not terminate

```
int maximum = Keyboard.readInt();
int counter = 1;
int number = 1;
while (counter <= maximum) {
    System.out.println(number);
    number = number + 2;
    counter--;
}
```

Loops

- Will this terminate?

```
int i;
i = 1;
while (i != 10) {
  //...
  i = i + 2;
}
```

Loops

- Will this terminate?

```
int i;
i = 100;
while (i != 0) {
  //...
  i = i / 2;
}
```

Loops

- Will this terminate?

```
int i;
i = 10;
while (i != 3) {
  //...
  i = i / 2;
}
```

Loops

- Will this terminate?

```
float i;
i = 10;
while (i != 0) {
  //...
  i = i / 2;
}
```

Loops

- Termination is important

Counting occurrences

- Problem: count the number of times that the letter 'e' occurs in a given string.

Counting occurrences

- Problem: count the number of times that the letter 'e' occurs in a given string.

- Analysis:
  - Input: a string s
  - Output: a positive integer n, equal to the number of times 'e' appears in s
  - Assumptions: s is all lowercase

- Design:
  - General idea: traverse s from left to right, and each time an 'e' is found, increment a counter.
  - Algorithm:

```
1. Set *counter* to 0
2. Set *index* to 0
3. While *index* < length of *s*, repeat:
   (a) Let *c* be the character at position *index* of *s*
   (b) If *c* is 'e', increment the *counter* by 1
   (c) Increment the *index* by 1
```

Counting occurrences

```
String s;
int counter, index;
s = Keyboard.readString();
counter = 0;
index = 0;
while (index < s.length()) {
  c = s.charAt(index);
  if (c == ‘e’) counter++;
  index++;
}
```

Abstraction

- The above algorithm does not change if instead of 'e', we count the occurrences of any letter x.

```
char x;
String s;
int counter, index;
s = Keyboard.readString();
x = Keyboard.readChar();
counter = 0;
index = 0;
while (index < s.length()) {
  c = s.charAt(index);
  if (c == x) counter++;
  index++;
}
```

- This works for any value of x and any value of s

Sum

- Problem: compute the sum of the first *n* positive integers for a given positive integer *n*

- Analysis:

  - Input: *n*
  - Output: $\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + (n-1) + n$
  - Assumptions: $n \in \mathbb{N}$

- Design:

```
1. Set total to 0

2. Set i to 1

3. While i <= n, repeat:

   (a) Set total to total + i
   (b) Set i to i+1
```

Sum

```
int n, i, total;
n = Keyboard.readInt();
i = 0;
total = 0;
while (i <= n) {
  total = total + i;
  i = i + 1;
}
System.out.println(total);
```

Product

```
int n, i, total;
n = Keyboard.readInt();
i = 0;
total = 0;
while (i <= n) {
  total = total + n;
  i = i + 1;
}
System.out.println(total);
```

This computes

$$\sum_{i=1}^{n} n = n^2$$

Factorial

- Problem: compute the product of the first *n* positive integers for a given positive integer *n*, i.e. the *factorial* of *n*

- Analysis:
  - Input: *n*
  - Output: $n! = \prod_{i=1}^{n} i = 1 \cdot 2 \cdot 3 \cdot ... \cdot (n-1) \cdot n$
  - Assumptions: $n \in \mathbb{N}$

- Design:

```
1. Set total to 0

2. Set i to 1

3. While i <= n, repeat:

  (a) Set total to total × i
  (b) Set i to i+1
```

Factorial

```
int n, i, total;
n = Keyboard.readInt();
i = 0;
total = 0;
while (i <= n) {
  total = total * i;
  i = i + 1;
}
System.out.println(total);
```

Guessing game

```
import cs1.Keyboard;
public class GuessingGame {
  public static void main(String[] args)
  {
    int die, guess, points, game;
    final int ROUNDS = 10;

    points = 0;
    game = 1;
    while (game <= ROUNDS) {
      System.out.print(“What is your guess? ”);
      guess = Keyboard.readInt();

      die = (int)(Math.random() * 6 + 1);
      if (guess == die) {
        points++;
      }
      game++;
    }
    System.out.println(“You guessed ”+points+“ tim
  }
}
```

Reverse

- Problem: Given any string, print the string in reverse.
- Analysis:
  - Information involved: a four letter word, *w*.
  - Input: *w*
  - Output: a word *v* which is the reverse of *w*
  - Definitions:
    - The *reverse* of a word *w* is a word *v* which has the the same characters as *w*, but in inverse order: the first letter of *v* is the last of *w*, the second letter of *v* is the second-to-last of *w*, etc.
  - Note: no restrictions on the string!

Design

The design for only strings of size 4:

1. Obtain the word *w*
2. Create a new word *v*, initially empty
3. Add the last character of *w* to the end of *v*
4. Add the third character of *w* to the end of *v*
5. Add the second character of *w* to the end of *v*
6. Add the first character of *w* to the end of v
7. Print *v*

Design

Generalise the design:

1. Create a new word *v*, initially empty
2. Add the last character of *w* to the end of *v*
3. Add the second to last character of *w* to the end of *v*
4. ...
5. Add the second character of *w* to the end of *v*
6. Add the first character of *w* to the end of v
7. Print *v*

Design

Generalise the design:

1. Create a new word *v*, initially empty
2. Traverse the string *w* from last character to first, adding the corresponding character at the end of *v*
3. Print *v*

Design

Generalise the design:

1. Create a new word *v*, initially empty

2. Set a variable *index* to be the last index of *w*

3. While the *index* is larger or equal to 0, repeat:

   (a) Let *c* be the character at index, of the string w.
   (b) Append *c* to *v*
   (c) decrement *index* by 1

4. Print *v*

Implementation

```
// This solution traverses w from right to left
String w, v;
int index;
char c;

v = ‘‘’’;
index = w.length() - 1;
while (index >= 0) {
  c = w.charAt(index);
  v = v + c;
  index--;
}
```

Implementation

```
// This solution traverses w from left to right
String w, v;
int index;
char c;

v = ‘‘’’;
index = 0;
while (index <= w.length() - 1) {
  c = w.charAt(index);
  v = ‘‘’’ + c + v;
  index++;
}
```

Prime numbers

- Problem: determine whether a given positive integer is prime or not
- Analysis:
  - Input: an integer n
  - Output: a boolean: true if n is prime, false otherwise
  - Definitions:
    - A *prime* number is a number which is divisible only by 1 and itself
    - An integer *a* is *divisible* by *b* if there is an integer k such that $a = kb$
  - Assumptions: *n* is positive

Prime numbers

- Basic idea: try to find a factor of n (i.e. a number that divides n), between 1 and n. If such number exists. then n is not prime, otherwise it is prime.

```
1. Set $is\_prime$ to true
2. Set $i$ to be 2
3. While $i < n$, repeat:
   (a) if $i$ divides $n$, then set $is\_prime$ to false
   (b) increment i by 1
4. Return the value of $is\_prime$
```

Prime numbers

```
boolean is_prime = true;
int i = 2;
while (i < n) {
  if (n % i == 0) is_prime = false;
  i++;
}
```

Prime numbers

```
boolean is_prime = true;
int i = 2;
while (i < n) {
  if (n % i == 0) {
    is_prime = false;
    i = n;
  }
  i++;
}
```

Prime numbers

```
boolean is_prime = true;
int i = 2;
while (i < n) {
  if (n % i == 0) {
    is_prime = false;
    break;
  }
  i++;
}
```

The end