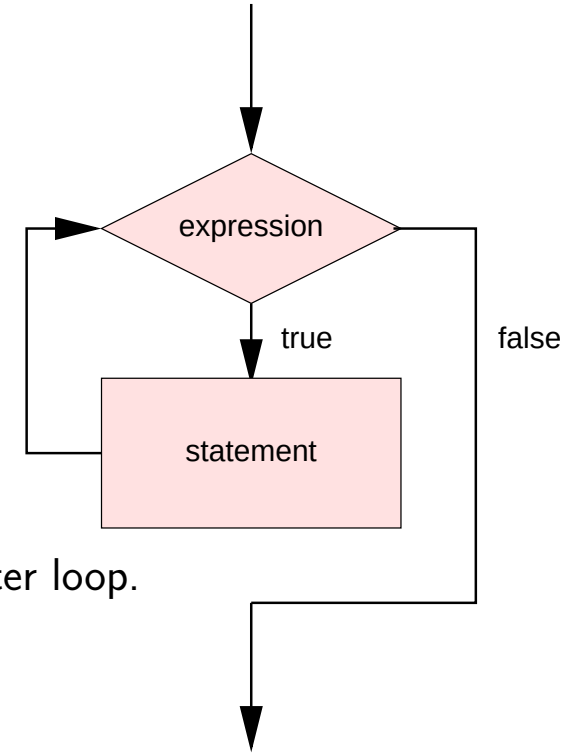
Control Structures: Loops

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

The `while` Statement

```
while (boolean expression)
  statement
```



- Unbounded loop
  - Boolean expression (the loop condition) is evaluated.
  - If result is true, the statement (the loop body) is executed.
  - If result is false, execution continues with first statement after loop.
  - Processes is repeated until loop condition becomes false.

Programmer must ensure that loop condition eventually becomes false.

Finite Loop

```
int limit = 3;
int i = 0;
int s = 0;
while (i < limit)
{
  s = s+i;
  i = i+1;
}
```

Loop body is executed three times.

Finite Loop

```
int n = 8;
int i = 0;
while (n % 2 == 0)
{
  n = n/2;
  i = i+1;
}
```

Loop body is executed three times.

Finite Loop

```
int n = 7;
int i = 0;
while (n % 2 == 0)
{
  n = n/2;
  i = i+1;
}
```

Loop body is executed zero times.

Infinite Loop

```
int n = 0;
int i = 0;
while (n % 2 == 0)
{
  n = n/2;
  i = i+1;
}
```

Loop body is executed an infinite number of times.

Assertions

```
while (expression)
{
  assert expression;
  ...
}
assert !expression;
```

- Loop condition holds at begin of every iteration.
- Loop condition does not hold after the loop.

Automatic knowledge about loop.

Loop Invariants

```
while (expression)
{
  assert expression && invariant;
  ...
}
assert !expression && invariant;
```

- Invariant holds at begin of every iteration.
- Invariant holds after the loop.

Knowledge about loop provided by programmer.

Example: Exponentiation

$r := a^b$:

```
int r = 1;
int i = 0;
while (i < b)
{
  r = r*a;
  i = i+1;
}
```

- In iteration i, we have $r = a^i$ and $i \leq b$.
- After the loop, we have $i \geq b$; this implies $i = b$ and $r = a^b$.

Algorithmic idea for program.

Assertions

`power(a,i)` denotes a^i.

```
int r = 1;
int i = 0;
while (i < b)
{
  assert i < b && r == power(a,i) && i <= b;
  r = r*a;
  i = i+1;
}
assert i >= b && r == power(a,i) && i <= b;
```

Can express algorithmic idea by program assertions.

More Loop Control

`break`

- Terminates enclosing looop.
- Continues with the first statement after the loop.

`continue`

- Skips the rest of the loop body.
- Continues with the evaluation of the loop condition and possibly the next loop iteration.

Example: Input Check

```
int i = 0;
int r = 1;
while (i < n)
{
  int a = Input.readInt();
  if (!Input.isOkay()) break;
  if (a < 0) continue;
  r = r*a;
  i = i+1;
}
```

Premature loop termination and skip over rest of loop body.

Example: Avoiding Extra Loop Control

```
int i = 0;
int r = 1;
boolean okay = true;
while (i < n && okay)
{
  int a = Input.readInt();
  okay = Input.isOkay();
  if (okay && a >= 0)
  {
    r = r*a;
    i = i+1;
  }
}
```

`break` and `continue` are convenient but not necessary.

Example: Stream Processing

```
while (true)
{
  int i = Input.readInt();
  if (Input.hasEnded()) break;
  if (!Input.isOkay())
  {
    String error = Input.getError();
    handle error case
    Input.clearError();
    continue;
  }
  handle input i
}
handle case that input stream has ended
```

Typical form of processing stream of input values.

Example: Interactive Input

```
int i;
while(true)
{
  System.out.print("Enter input: ");
  i = Input.readInt();
  if (Input.isOkay() && other condition on i) break;
  if (Input.hasEnded())
  {
    System.out.println("End of input!");
    System.exit(0);
  }
  System.out.println("Invalid input!");
  Input.clearError();
}
handle input i
```

Typical form of reading input from human user.

Loop Control with Labels

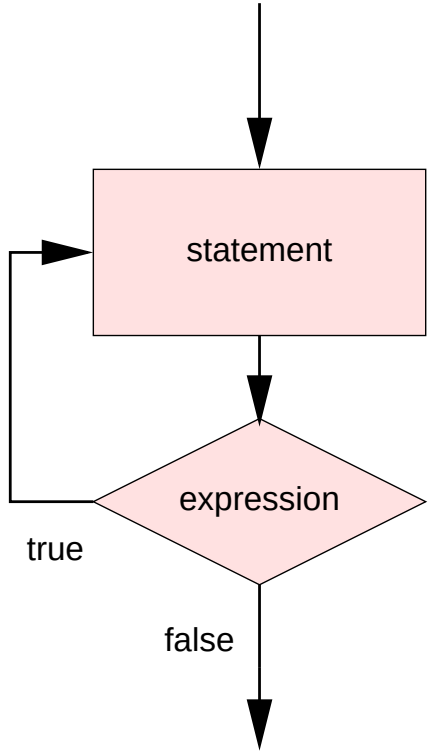
Nested Loops can be labelled.

```
m: while (...)
   {
     while (...)
     {
       if (...) break m;
       if (...) continue m;
     }
   }
```

May also break out of/continue with outer loop.

The `do` Loop

```
do
  statement
while (boolean expression)
```



- Unbounded loop
  - The statement (the loop body) is executed.
  - Boolean expression (the loop condition) is evaluated.
  - If result is true, the loop is executed again.
  - If result is false, execution continues with first statement after loop.

Loop body is executed at least once

Example

Read one number after another and print it.

```
int i;
do
{
  i = Input.readInt();
  if (!Input.isOkay()) break;
  System.out.println(i);
}
while (i != 0)
```

Loop terminates when value 0 has been read.

`do` **versus** `while`

Which loop form is more powerful?

- Effect of a `do` loop can be expressed with `while`:

```
statement
while (expression)
  statement
```

- Effect of a `while` loop can be expressed with `do`:

```
if (expression)
  do
    statement
  while (expression)
```

Both loops have equal power.

Assertions

```
do
{
  assert invariant;
  statement;
}
while (expression);
assert !expression && invariant;
```

- Loop condition does not hold after the loop.
- Invariant must hold at begin of every iteration and at end of loop.

Example: Printing a Number

```
int number = i;
String reverse = "";
do
{
  int digit = number%10;
  reverse = reverse + digit;
  number = number/10;
}
while (number > 0);
System.out.println(reverse);
```

Takes number *i* (say 2846) and prints its reverse (6482).

Variable Values

`number`	`reverse`
2846	
284	6
28	64
2	648
0	6482

- "2846" equals "28" + rev("64").
- "2846" equals "2" + rev("648").
- ...

Loop Annotation

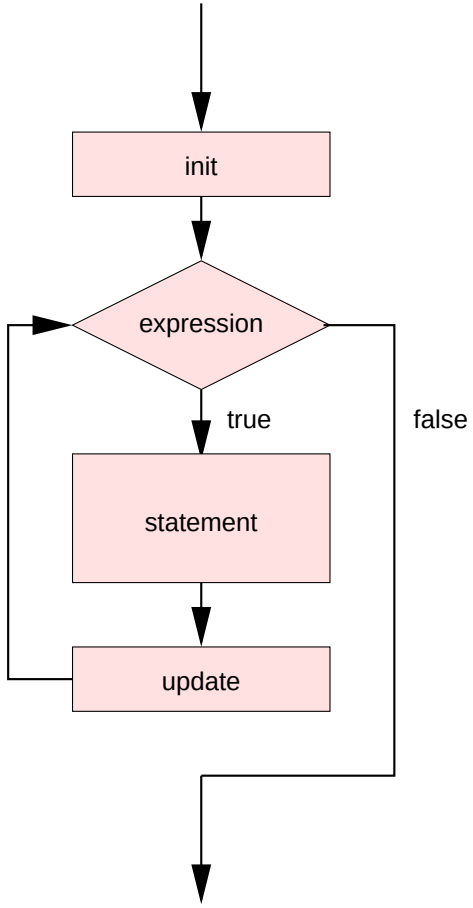
```
do
{
  assert str(i).equals(str(number)+rev(reverse));
  int digit = number%10;
  reverse = (reverse*10) + digit;
  number = number/10;
}
while (number > 0);
assert !(number > 0) && str(i).equals(str(number)+rev(reverse));
```

Assertions expressed with the help of some pseudo-functions:

- `str` returns string representation of a number (“” for 0).
- `rev` returns the reverse of a string.

The `for` Loop

```
for (init; boolean expression; update)
  statement
```



- Shortcut for loop pattern:

```
{
  init;
  while (boolean expression)
  {
    statement;
    update;
  }
}
```

Frequently used for bounded loops.

Increment Operators

- The increment operator

  ```
  var++
  ```

  executes

  ```
  var = var+1
  ```

- The increment operator

  ```
  var += exp
  ```

  executes

  ```
  var = var+exp
  ```

Shortcuts for frequently occuring assignments.

Bounded Loops

```
for (int var = val ; var < exp ; var++)
  statement
```

- Initialization "`int var = ...`"
  - Variable *var* is declared local to loop.
  - Initialization "`var = ...`" uses variable declared outside of loop.
- Loop is executed *exp* − *val* times:

  var=*val*
  var=*val*+1
  ...
  var=*exp*−1.

Bounded number of iterations.

Example

- Program:

```
for (int i = 0; i < n; i++)
  System.out.println("i: " + i);
```

- Output:

```
i: 0
i: 1
i: 2
...
i: n-1
```

- $n = 0$: no value is printed.

Example

- Program:

```
for (int i = 0; i < n; i++)
  System.out.println("i: " + i);
for (int i = 0; i < n; i++)
  System.out.println("i: " + (-i));
```

- Output:

```
i: 0
...
i: n-1
i: 0
i: -1
i: -2
...
i: -(n-1)
```

Decrement Operators

- The decrement operator

  ```
  var--
  ```

  executes

  ```
  var = var-1
  ```

- The decrement operator

  ```
  var -= exp
  ```

  executes

  ```
  var = var-exp
  ```

Shortcuts for frequently occuring assignments.

Decreasing Loops

```
for (int var = val; var >= exp; var--)
  statement
```

- Program:

```
for (int i = n-1; i >= 0; i--)
  System.out.println("i: " + i);
```

- Output:

```
n-1
...
2
1
0
```

Bounded number of iterations.

Assertions

```
for (int var = init; var < max; var++)
{
  assert init <= var && var < max && invariant;
  statement
}
assert invariant[var ← max];
```

- Loop index is in denoted range withing loop.
- Invariant must hold at begin of every loop.
- Invariant must hold for $var = max$ after loop.

Express loop idea by invariant.

Example: Exponentiation

$r := a^b$.

```
int r = 1;
for (int i = 0; i < b; i++)
  r = r*a;
```

Annotation:

```
int r = 1;
for (int i = 0; i < b; i++)
{
  assert 0 <= i && i < b && r = exp(a, i);
  r = r*a;
}
assert r = exp(a, b);
```