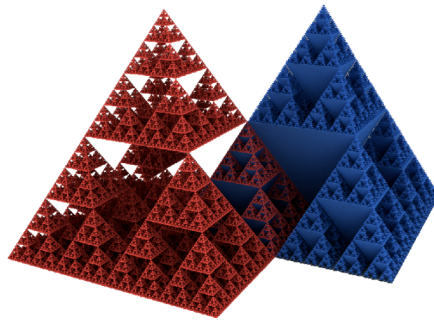


COMPSCI 101

Principles of Programming

Lecture 13

While & for loops (chapter 13)
Break & Continue statements (extra)



Review

Return

Returning values

- Allows a method to pass information back to the invoking code
- Can only **return** a single value (primitive or object)
- Method is executed then the return value **is used in place of** the method call

Formal Syntax

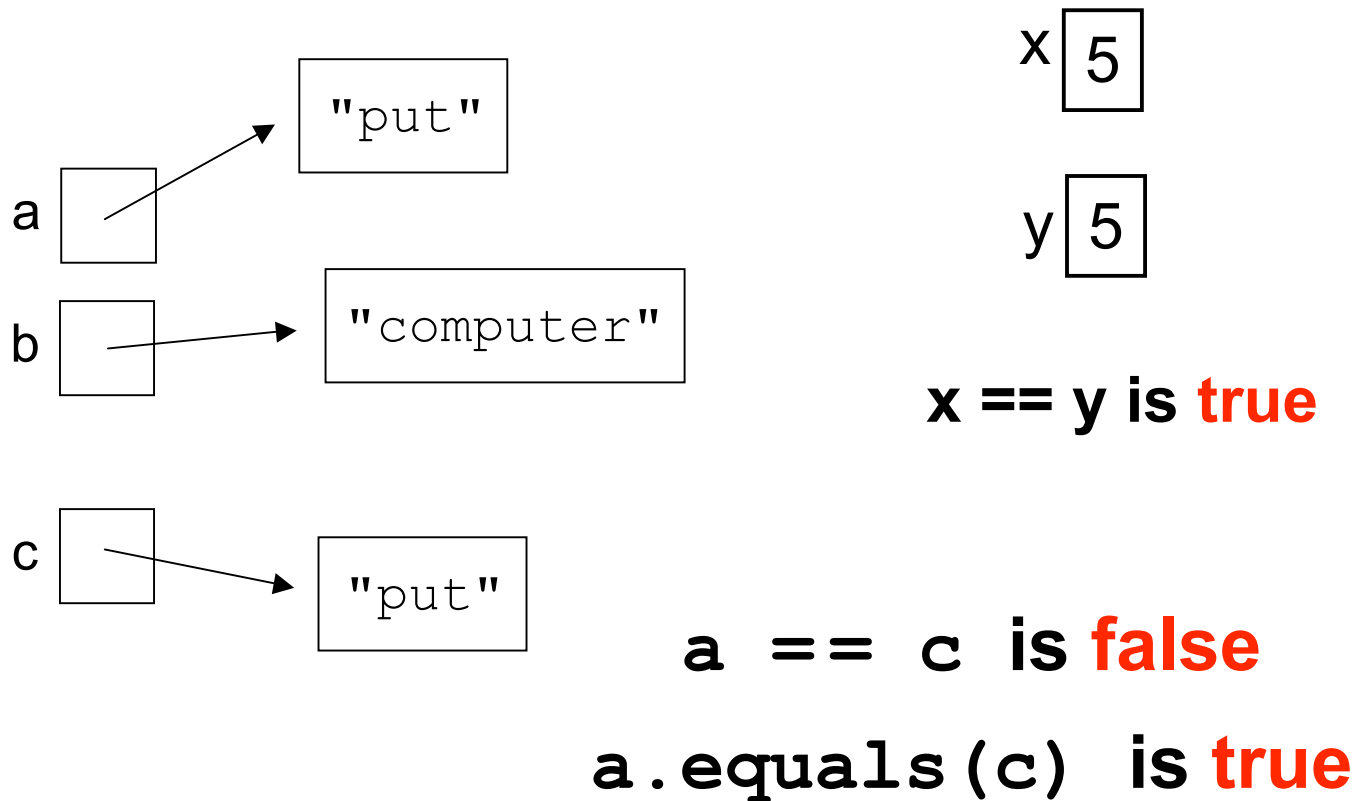
```
private <returnType> methodIdentifier( ... parameters ... ) {  
    .. code goes here  
    return <value>;  
}
```

*the value returned must be the **same type** as the one declared in the method header*

```
private String getName() {  
    return "Fred Fish";  
}
```

Comparing Strings

To compare two String variables character by character, we must use the `.equals()` instance method



Looping

Repeating a task

We can use a *loop* to repeatedly execute a statement (or a block of statements).

Formal Syntax

The syntax of a **while** loop is similar to that of an **if** statement:

```
while( condition ) {  
    statement_1;  
    ...  
    statement_n;  
}
```

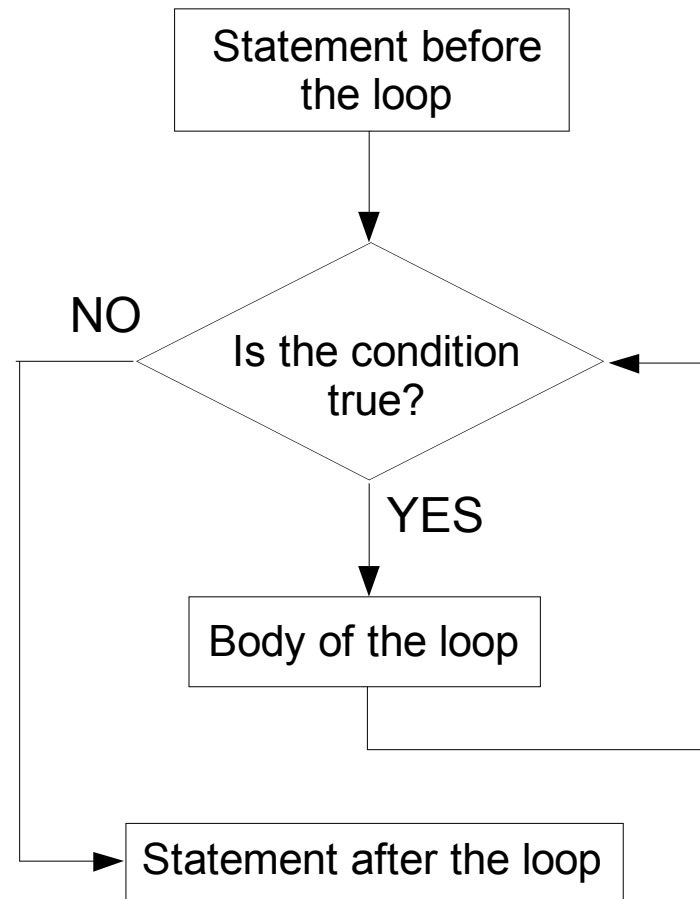
While loop

The syntax of a **while** loop is similar to that of an **if** statement:

```
while( condition ) {  
    statement_1;  
    ...  
    statement_n;  
}
```

- First, the condition is tested.
- If the condition is true, then the statements (known as the *body* of the loop) are executed.
- After the statements are executed, control returns to the top of the loop, and the condition is tested again.
- As long as the condition is true, the statements will be executed

Flow of control



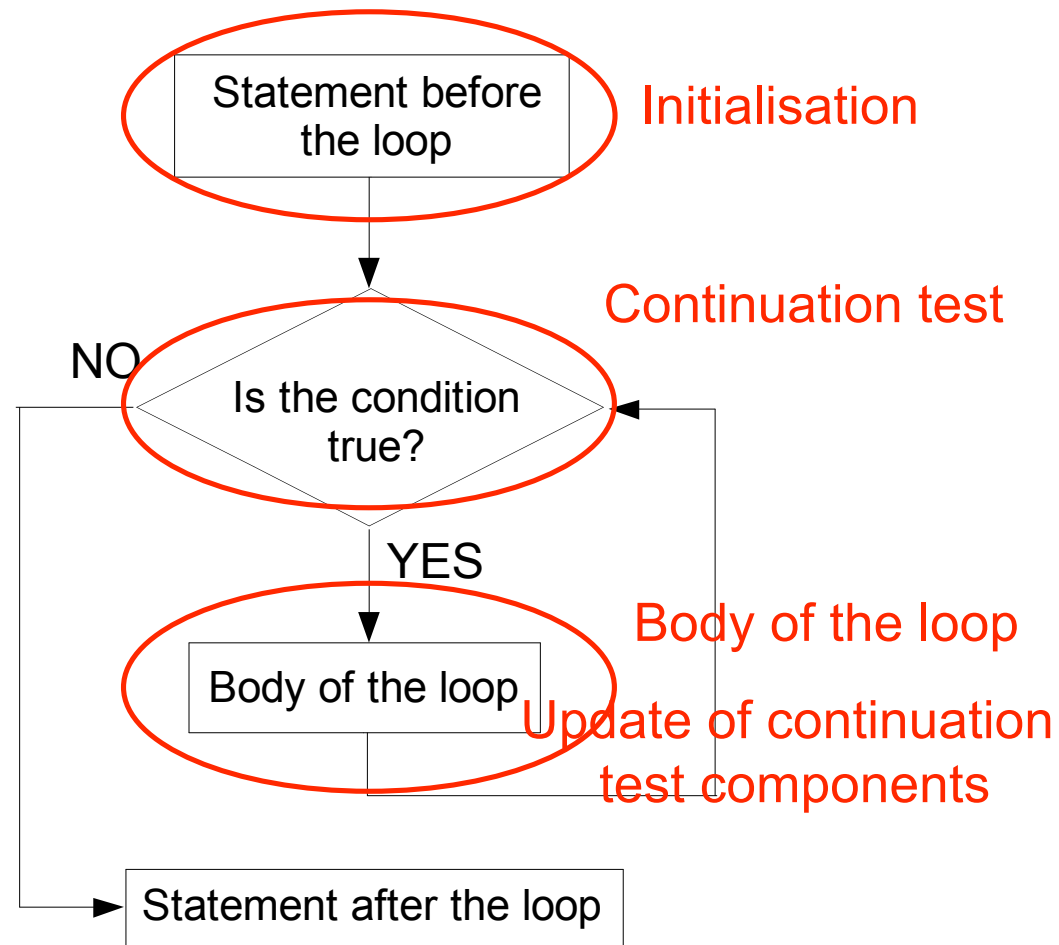
Example

For example, the following program simulates paying for parking:

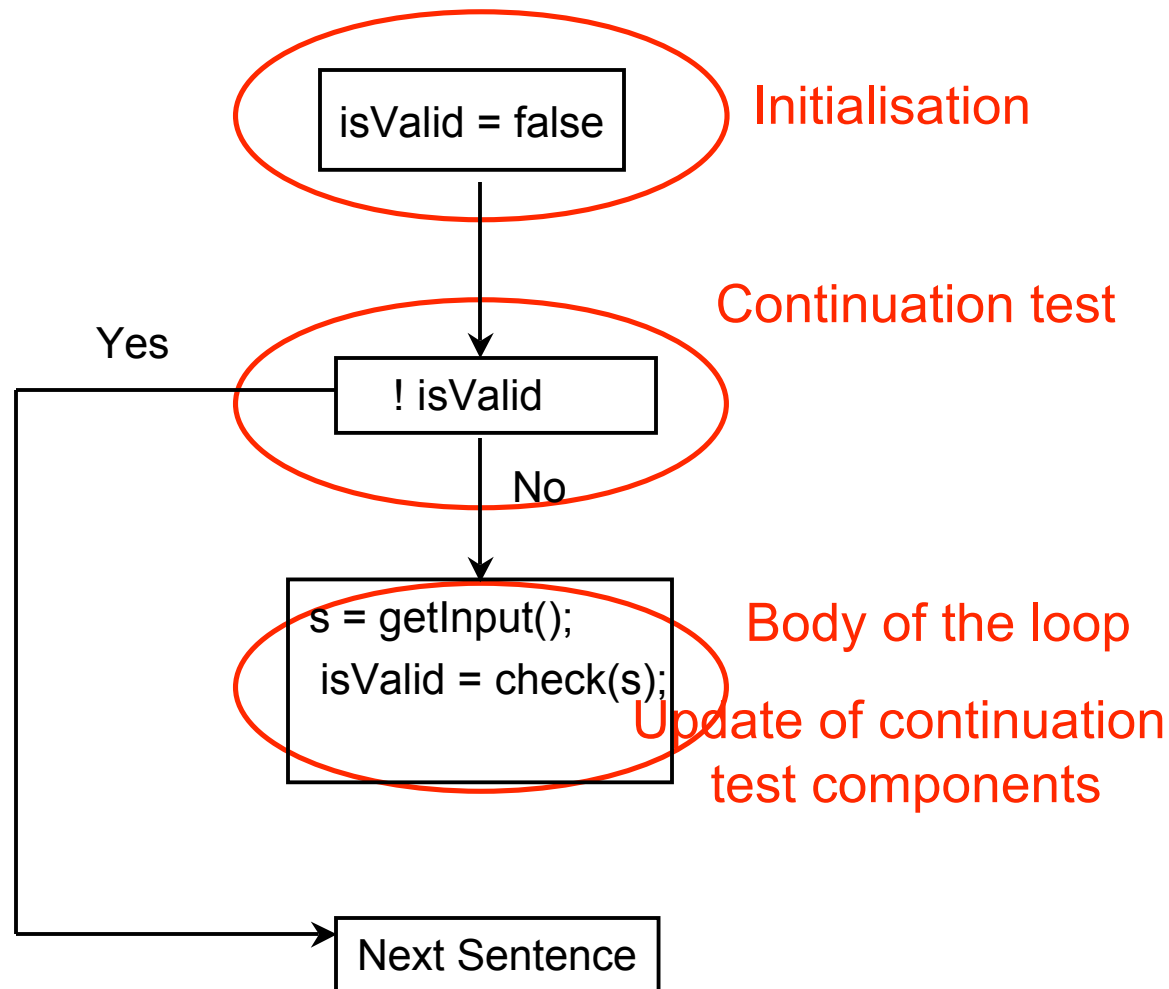
```
int numberOfCoins = 4;  
while( numberOfCoins > 0 ) {  
    numberOfCoins--;  
    System.out.println("Insert a coin");  
}
```

```
Insert a coin  
Insert a coin  
Insert a coin  
Insert a coin
```

Designing a loop



Designing a loop example

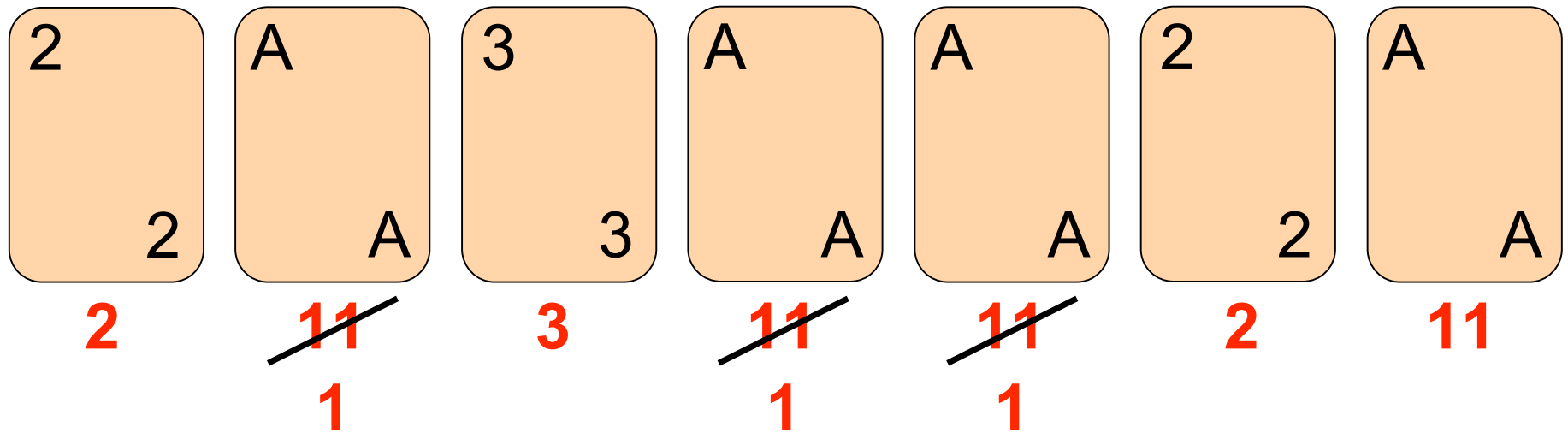


Common situations

Loops commonly occur in situations such as:

- validating input
- processing until done - hammering a nail
- counter - dealing cards
- reading an unspecified number of values - paying bills
- iterating through a data structure - adding arrays

Aces



~~Total = 51~~
~~= 41~~
~~= 31~~
= 21

```
int total = 51;  
int numberOfAces = 4;
```

Aces

```
if (total > 21) && (numberOfAces > 0) {  
    total = total - 10;  
    numberOfAces--;  
}
```

*one way we could calculate the
total of the hand would be to use
several if statements*

```
if (total > 21) && (numberOfAces > 0) {  
    total = total - 10;  
    numberOfAces--;  
}  
  
....
```

```
while (total > 21) && (numberOfAces > 0) {  
    total = total - 10;  
    numberOfAces--;  
}
```

*another way would be
to use a loop*

Blocks of code – variable scope

Variables only exist in the block of code in which they are declared

```
public class LectureExample{
    public void start() {
        double cost = 53.0; //Cost in dollars
        final double GST = 0.125; //Tax percentage

        int age = (int) (Math.random() * 100 + 1);
        if (age <= 12 || age >= 65){
            double discount = 0.15;
            System.out.println("Eligible for discount");
        }

        double price = cost * (GST + 1) * (1 - discount);
        System.out.println(price);
    }
}
```

scope

illegal

For loop

A **for loop** is another type of loop:

- more compact than a while loop
- otherwise equivalent

Formal Syntax

```
for( initialisation; condition; update ) {  
    statement_1;  
    ...  
    statement_n;  
}
```

Comparison of while and for loops

```
int counter = 0;
while (counter < MAX) {
    System.out.println(counter);
    counter++;
}
```

```
for (int counter = 0; counter < MAX; counter++) {
    System.out.println(counter);
}
```

Scope of variables

Any variables declared in the initialization section of the `for` loop only exist inside the `for` loop – and cannot be referred to outside of the loop.

Any variables declared inside the body of a loop are also local to the loop body only

```
for (int i = 0; i < 10; i++) {  
    int count = 0;  
    count = count + i;  
    System.out.println(count);  
}
```

- the scope of both `i` and `count` is *inside* the loop body
- they cannot be referred to by code outside the body of the loop

**The following material
is not in the book.**

Breaking Out of the Loop

In both the *while* and the *for* statements, the test to stop looping has to be at the start of the loop, but often the test comes naturally in the middle. Happily, Java provides a way of leaving loops early: the *break* statement.

“*break*” is used as follows:

```
<loop header> {  
    ... <code>  
    if (test2) break; // exit from while loop  
    ... <code skipped if “break” executed>  
}
```

The “*break*” statement causes us to jump out of the loop completely!

Break Example

For example:

```
inputsValid = false;
while (! inputsValid) {
    System.out.println("Type in command");
    s = Keyboard.readInput();
    inputsValid = isValid(s);
    if (! inputsValid) System.out.println("Bad command");
}
```

Could be re-written as:

```
while (true) {
    System.out.println("Type in command");
    s = Keyboard.readInput();
    if (isValid(s)) break;
    System.out.println("Bad command");
}
```

Skipping the Rest of One Iteration

Sometimes however, we don't want to break out of the loop, we simply want to skip the rest of the current iteration. For example:

```
while(test1) {  
    <code to get some input>  
    if (! inputIsComment) {  
        <code to process the input>  
    }  
}
```

If the input is just a comment, i.e., doesn't need processing, then we want to skip all the processing code and loop back to getting more input and processing that.

To skip the rest of the current iteration, we use the “*continue*” statement.

Continue Looping

For example,

```
while(test1) {  
    <code to get some input>  
    if (! inputIsComment) {  
        <code to process the input>  
    }  
}
```

could be re-written as:

```
while(test1) {  
    <code to get some input>  
    if (inputIsComment) continue;  
    <code to process the input>  
}
```

Summary

Looping is a common program activity. Loops normally can be decomposed into four parts:

- continuation test,
- body,
- updating the continuation test components,
- initialisation.

We have seen how to use both *while* and *for* loops.

We have also seen how to use *break* statements to “exit” loop statements and to use *continue* statements to “skip” the rest of the current iteration of the loop and “continue” on to the next iteration.