

Software Development II

Lecture 3- Loops and input validation

Read Chapters 1 and 2 - Java for Everyone.

From Last Week

Data types

- Variables
 - Declaration
 - Strings
 - Booleans
 - Operators
 - Casting
- Constants

Control structures

- Conditionals
 - If
 - If-else
 - Nested if
 - If-else ladder
 - Switch Case

This week

Loops

- For loop
- While loop
- Do .. While loop

Jump statements

- Break
- Continue

Data input validation

- Ranges
- Data types
- Required fields

Debugging

- Debugging strategies

Why Loops?

Imagine we want to print all the numbers from 1 to 100

- We could do something like this:

```
System.out.println("1");  
System.out.println("2");  
System.out.println("3");  
System.out.println("4");  
System.out.println("5");  
System.out.println("6");  
System.out.println("7");  
...  
System.out.println("100");
```

This task requires 100 lines of code.

If we want to modify and print the numbers in another format such as:

1 – 2 – 3

We would have to modify 100 lines of code.

Is there a better way?

Loops

Loops are a fundamental concept in programming that allow to **repeat a set of instructions multiple times**. They allow us to automate repetitive tasks efficiently, so we do not have to perform the same operation multiple times.

Loops

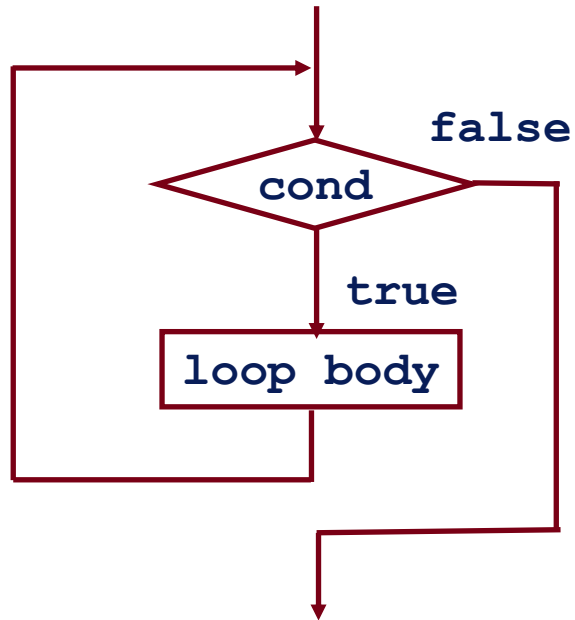
- Two kinds of loops
 - **Finite loop**: The statements in the block may be executed a fix number of times, from zero.
 - **Infinite loop**: A loop that continues forever.
- A loop consist of;
 - Body of the loop
 - Control statement
- Read more about Loop types in java
 - <https://techvidvan.com/tutorials/java-loops/>

Loops

- The **while** loop
- The **do while** loop
- The **for** loop

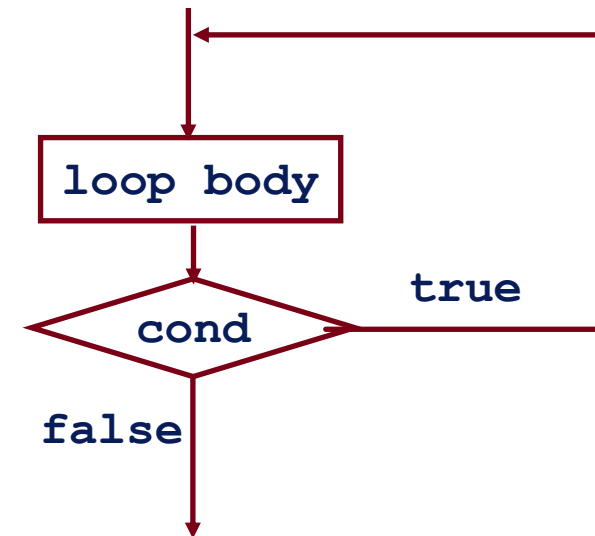
Loops

Entry Control loop



- If condition is not satisfied body of the loop is never executed.

Exit Control loop



- The body of the loop is executed unconditionally for the 1st time.

The `while` Loop

```
initialization;  
while (condition) {  
    statement1;  
    statement2;  
    ...  
    statementN;  
}
```

condition is any logical expression, as in if

The body of the loop

- This is an **entry-controlled** loop

- To execute a block of code until a conditions is met (when you **do not** know **exactly** how many times).

The condition is a boolean expression that determines whether the code inside the while loop should be executed

```
while (condition)
{
    // your code here
}
```

```
System.out.println("Enter a number smaller than 10:");
Scanner getInput = new Scanner(System.in);
int number = getInput.nextInt();

while (number < 10 )
{
    System.out.println(number);
    number++;
}
```

Input: **5**

Output:

5

6

7

8

9

Use of While Loop

- Example: We ask the user to enter a value until a valid input is provided

```
String password = "JKLSxd$";
System.out.println("Enter a password:");
Scanner getInput = new Scanner(System.in);
String user_password = getInput.next();

while (!password.equals(user_password)) {
    System.out.println("Password is incorrect. Please enter a password:");
    user_password = getInput.next();
}
```

- we don't know how many times the user will input the password incorrectly.
- If the password is correct the first time, the while loop code is not executed.

The `while` Loop contd...

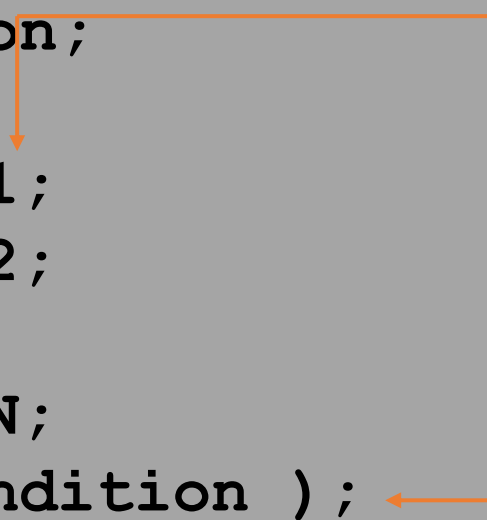
- **Initialization**: The variables tested in the condition must be initialized to some values. If the condition is false at the outset, the loop is never entered.
- **Testing**: The condition is tested before each iteration. If false, the program continues with the first statement after the loop.
- **Change**: At least one of the variables tested in the condition must change within the body of the loop.

Try this...

- Write a program that uses a **while** loop to print the sum of integers from 1 to 10.

The **do-while** Loop

```
initialization;  
do {  
    statement1;  
    statement2;  
    ...  
    statementN;  
} while ( condition );
```



The code runs through the body of the loop at least once

if condition is false, the next iteration is not executed

- This is an **exit-controlled** loop

Try this...

- Write a program that uses **do-while** loop to print the sum of squares of integers from 1 to 10.

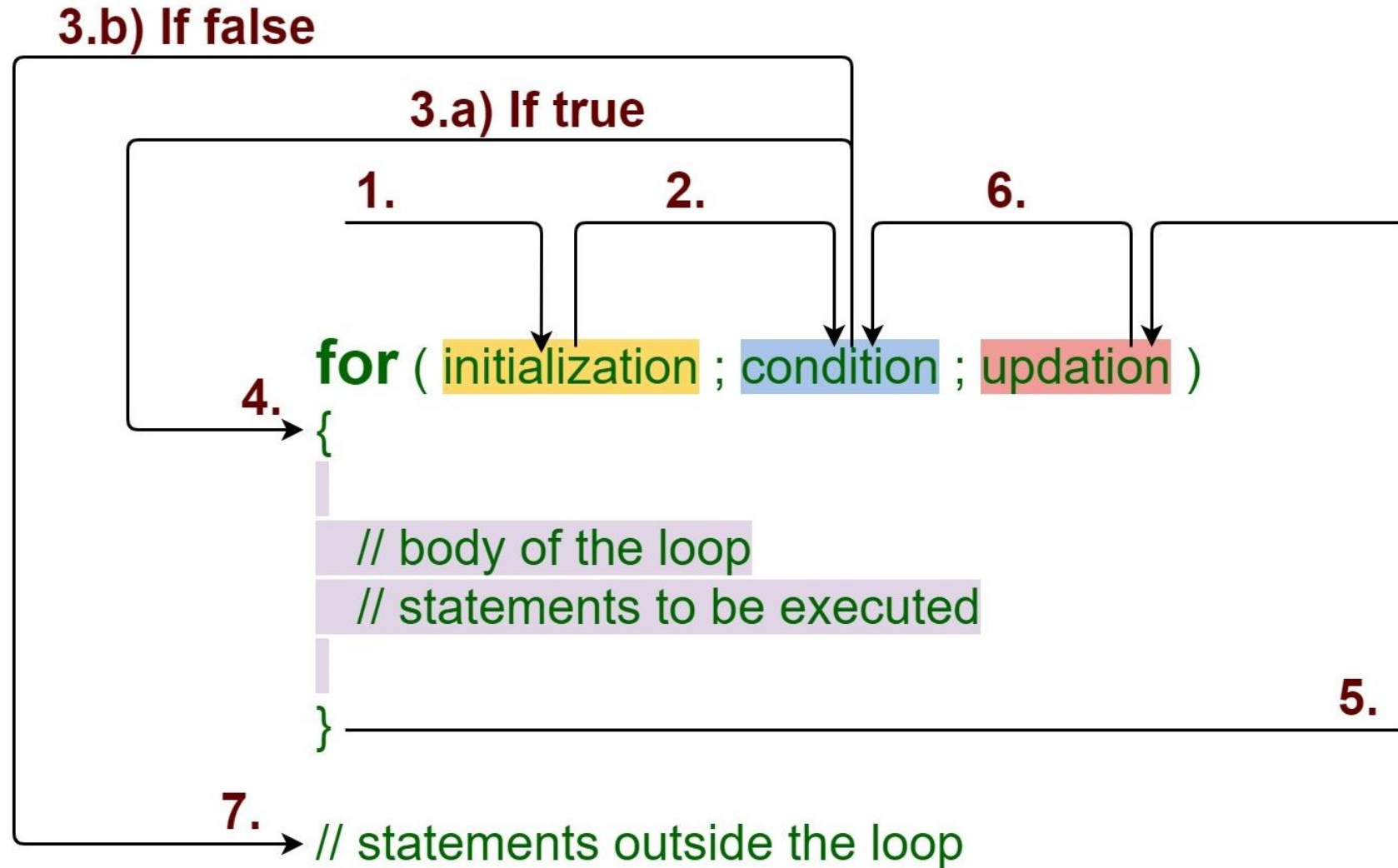
The `for` Loop

- `for` is a shorthand that combines in one statement initialization, testing, and change:

```
for (initialization; condition; change )  
{  
    statement1;  
    statement2;  
    ...  
    statementN;  
}
```

- This is an **entry-controlled** loop.
- Is it possible to have multiple conditions in a For loop?

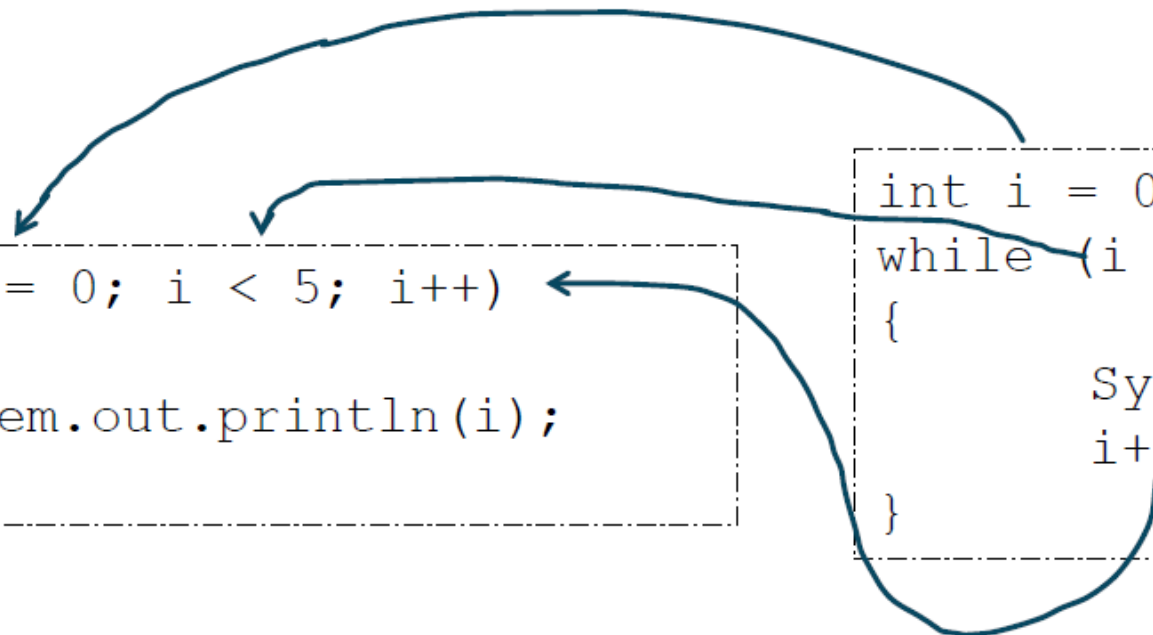
The `for` Loop – Order of Execution



While loop vs for loop

```
for (int i = 0; i < 5; i++)  
{  
    System.out.println(i);  
}
```

```
int i = 0;  
while (i < 5)  
{  
    System.out.println(i);  
    i++;  
}
```



The diagram consists of three hand-drawn blue arrows. The first arrow starts at the 'i++' part of the for loop's header and points to the 'i++;' statement inside the while loop's body. The second arrow starts at the 'i < 5' condition of the for loop and points to the 'while (i < 5)' condition of the while loop. The third arrow starts at the '{' opening brace of the for loop and points to the '{' opening brace of the while loop.

Try this...

1. Write a program that uses **for** loop to generate even numbers that are less than 10.

i.e. 0

2

4

6

8

Summary of Loop Control Structures

for

```
For
  (n=1;n<=10;n++)
  {
    .....
    .....
  }
```

while

```
n = 1;
while (n<=10)
{
  .....
  .....
  n = n+1;
}
```

do while

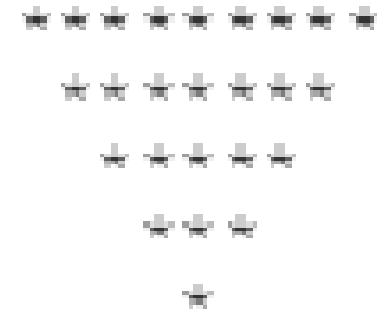
```
n = 1;
do
{
  .....
  .....
  n = n+1;
}
while (n<=10);
```

Nested Loops

- A **nested loop** is a looping construct that appears as one of the statements within the body of another loop construct.
- When using such a form;
 - Do need to take care that the conditions used to terminate each loop
 - do not interact in a destructive manner!

Try these...

- Write a program to print a multiplication table using **for** loops.
- Try to use nested for loops to print below patterns



Example: Loop & Switch: Menu in a loop

```
Scanner input = new Scanner(System.in);
int option;
boolean quit = false;
System.out.println("* MENU *");
System.out.println("1.- Print user name");
System.out.println("2.- Add user");
System.out.println("3.- Delete user");
System.out.println("4.- Quit");

while (!quit){
    System.out.print("Choose an option: ");
    option = input.nextInt();
    switch(option){
        case 1:
            // some code to print user name
            break;
        case 2:
            // some code to add a user
            break;
        case 3:
            // some code to delete a user
            break;
        case 4:
            quit = true;
            break;
        default:
            System.out.println("Incorrect option.");
    }
}
```

Selection of Loops

- Identify the **most suitable** type of loop for the use cases below.
 - You are the module leader of SD2 module and you have 550 students in your module. For each student, you want to calculate which is their average mark. Which looping structure is ideal?
 - You have been asked to implement a menu. The menu has several options which user can select from. Option 0 is to exist program. The user can select as many as options they want until they choose option 0. Which looping structure is ideal?

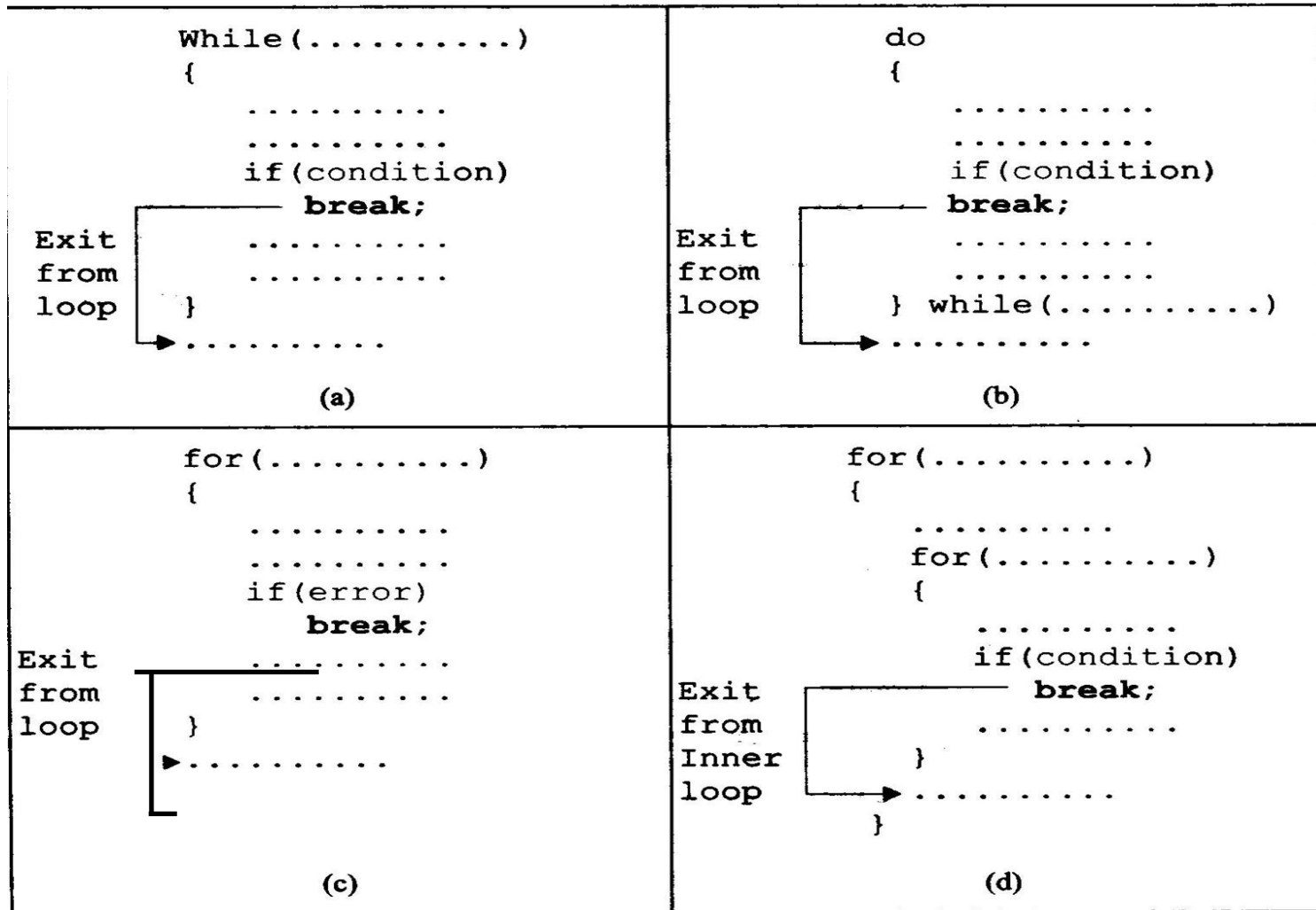
Jump Statements

- **break** – To jump out of a loop
- **continue** – To skip a part of a loop

break Statement

- Used to achieve an early exit from a loop.
- Used within **while**, **do while**, **for** and **switch**.
- **break** results an immediate **exit from the loop containing it** (nearest loop).
- **break** will exit only a single loop.

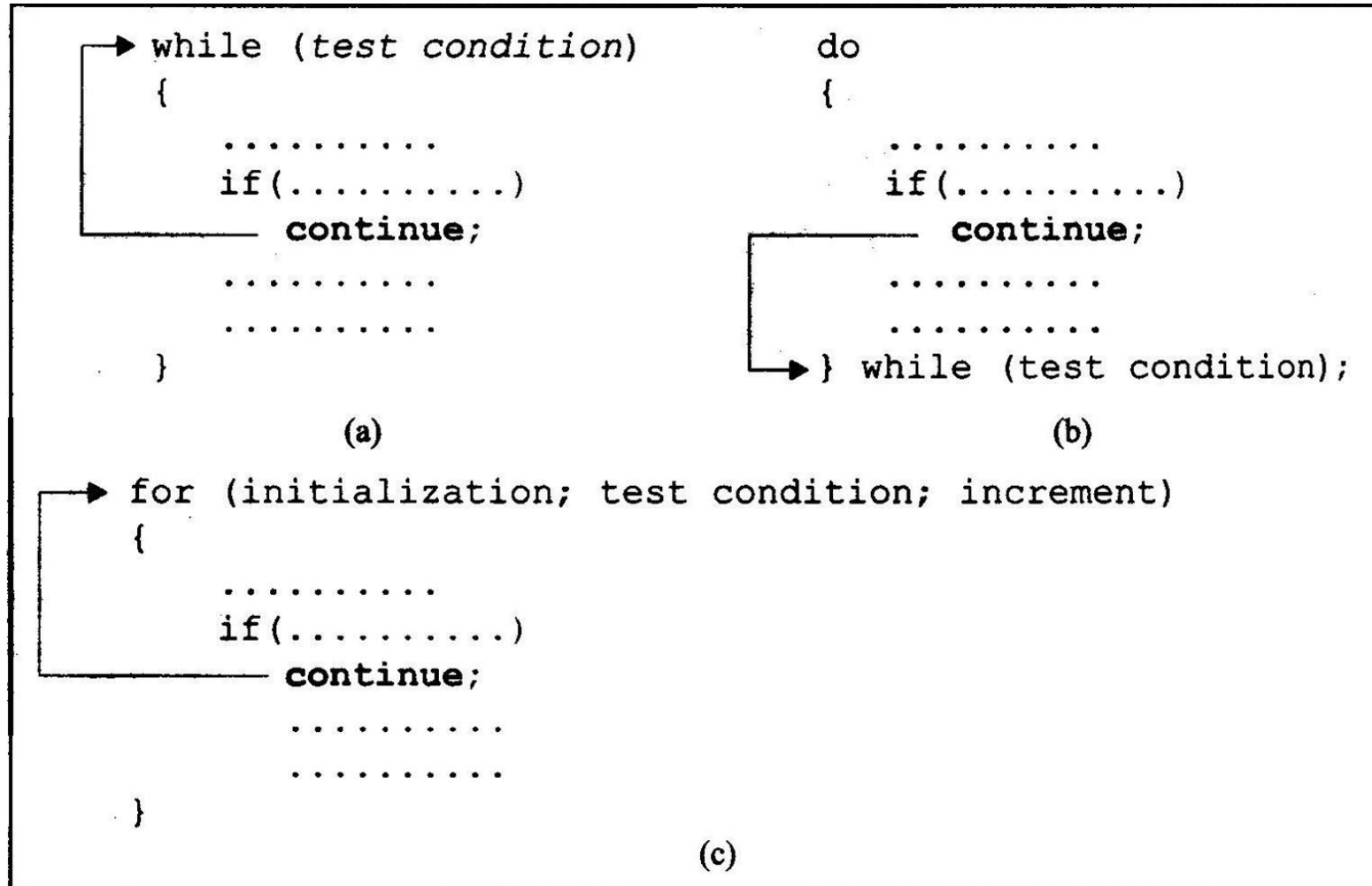
break Statement



continue Statement

- Restarts the current loop.
- Causes the loop to be **continues with the next iteration** after skipping any statements in between.
- Used within **while**, **do while** and **for**.

continue Statement



Common Issues

- **Incorrect Test Condition**
- Loop body only executes if the condition is TRUE
- If bal is initialized as less than the TARGET and should grow until it reaches TARGET, Which version is correct?

```
while(bal >= TARGET) {  
    year = year + 1  
    interest = bal * RATE  
    bal = bal + interest  
}
```

```
while(bal < TARGET) {  
    year = year + 1  
    interest = bal * RATE  
    bal = bal + interest  
}
```

Common Issues 2

- Infinite loops

```
counter = 1
while(counter != 100) {    // Runs forever
    print(counter)         // counter not updated
}
```

```
counter = 1
while (counter != 100) {    # counter increment 1,3,5,
    print(counter)         # 7,9,11.....99,101.
    counter = counter + 2
}
```

Common Issues 3

- **Off by One error/ how many times you want to run the loop**

```
counter = 0
while (counter < 10) {      // Q1.How many passes?
    counter = counter + 1
}
```

```
counter = 1
while (counter < 10) {      // Q2.How many passes?
    counter = counter + 1
}
```

```
counter = 0
while (counter <= 10) {     // Q3.How many passes?
    counter = counter + 1
}
```

```
counter = 1
while (counter <= 10) {     // Q4.How many passes?
    counter = counter + 1
}
```


Infinite Loops

- **While, do while and For comparison**

```
while (true) {  
    // do something  
}
```

```
for (;;) {  
    // do something  
}
```

```
do {  
    // do something  
} while (true);
```

Try this

- It is required to write a program that reads employee information iteratively. (Employee name, emp_no, job title)
 - Before reading the Employee number, make sure employee number is an integer number. Otherwise exit from the program with an appropriate error.
- Additionally users are allowed to decide the termination point.
- This means that the program prompts user for entering employee information.
- Soon after the input process the input values are displayed for the user.
- Then ask the user whether he wants to continue or exit. For example, you may ask users to enter 'Yes' to continue and 'No' to terminate.

Data Input Validation and Exception Handling

Data input validation

- Data input validation is a crucial for robust software development as it helps us to ensure that **the data entered by users meet the expected criteria**, preventing errors, security vulnerabilities and other issues.
- Sample Validation points
 - Ranges
 - Data Types
 - Required fields
 - Etc..

Range Validations

- Considering the following code:

```
Scanner input = new Scanner(System.in);  
System.out.println("Enter your age: ");  
int age = input.nextInt();
```

What happens when we enter -10?

```
Enter your age:
```

```
-10
```

```
-----
```

```
-----
```

```
BUILD SUCCESS
```

```
-----
```

```
-----
```

Is age -10 correct?

Range Validations

```
Scanner input = new Scanner(System.in);
boolean correct = false;
while (!correct) {
    System.out.println("Enter your age: ");
    int age = input.nextInt();
    if (age < 0 || age > 150) {
        System.out.println("Age must be between 0 and 150.");
    }
    else correct = true;
}
```

Enter your age:

-10

Age must be between 0 and 150.

Enter your age:

10

Data type Validation

- Considering the following code:

```
Scanner input = new Scanner(System.in);  
System.out.println("Enter your age: ");  
int age = input.nextInt();
```

What happens when we enter "twenty"?

Is expecting an int but we enter a String

Enter your age:
twenty

Exception in thread "main" java.util.InputMismatchException

```
at java.base/java.util.Scanner.throwFor(Scanner.java:939)  
at java.base/java.util.Scanner.next(Scanner.java:1594)  
at java.base/java.util.Scanner.nextInt(Scanner.java:2258)  
at java.base/java.util.Scanner.nextInt(Scanner.java:2212)  
at com.mycompany.mavenproject2.Mavenproject2.main(Mavenproject2.java:18)
```

Command execution failed.

Data type Validation

- Scanner provide the ability to look at the next token in the input stream before we actually read it into our program.
 - hasNextInt()
 - hasNextDouble()
 - hasNext()
 - etc...

```
if (s.hasNextInt())
{
    age = s.nextInt(); //read user input(age)
}
else
{
    age = 30; //assign a default value
    String junk = s.next();
}
```


Exercise

- Write a program to read the number of novels you have as an integer.
 - If it is greater than 20 – display “Wow!”
 - If it is less than or equals to 20 – display “Not Bad”
 - If it is equal to zero – display “Buy One Now!!”
- Validate the input for its data type. In case of a wrong data type, exit from the program.

Java Exception Handling

- When there is an error, the program will stop, and Java will print a message.
- With **try** and **catch** you can handle the errors (in a more controlled way).

```
try{  
    //code  
}  
catch (Exception e) {  
    System.out.println("Error message here");  
}
```

Data type Validation with try/catch

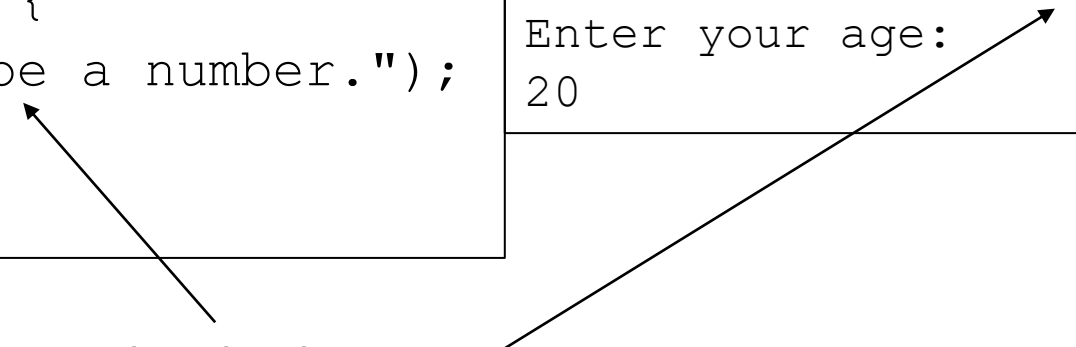
Import: `java.util.InputMismatchException;`

```
boolean correct = false;
while (!correct) {
    try {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter your age: ");
        int age = input.nextInt();
        correct = true;
    } catch (InputMismatchException e) {
        System.out.println("Age must be a number.");
    }
}
```

Output

```
Enter your age:
twenty
Age must be a number.
Enter your age:
20
```

Catch exception, and we let the user try again without stopping the program.



Data type Validation with try/catch

Solution: validate the range

```
boolean valid = false;
while (!valid) {
    try {
        Scanner getInput = new Scanner(System.in);
        System.out.println("Enter dividend: ");
        int dividend = getInput.nextInt();
        System.out.println("Enter divisor: ");
        int divisor = getInput.nextInt();
        double result = dividend / divisor;
        valid = true;
    } catch (ArithmeticException e) {
        System.out.println("Input not valid.");
    }
}
```

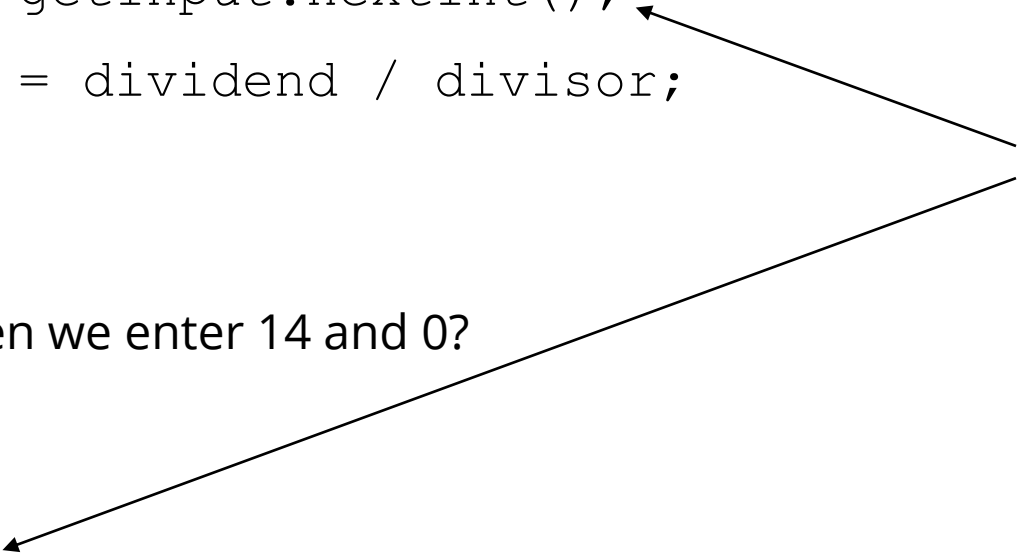
```
Enter dividend:
10
Enter divisor:
0
Input not valid.
Enter dividend:
10
Enter divisor:
5
```

Range validation with try/catch

- Considering the following code:

```
Scanner getInput = new Scanner(System.in);  
int dividend = getInput.nextInt();  
int divisor = getInput.nextInt();  
double result = dividend / divisor;
```

Both numbers are correct and have the correct type, however we cannot divide by 0



What happens when we enter 14 and 0?

```
Enter dividend:  
14  
Enter divisor:  
0
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
at com.mycompany.mavenproject2.Mavenproject2.main(Mavenproject2.java:24)  
Command execution failed.
```

Range validation with try/catch

Solution: validate the range

```
boolean valid = false;
while (!valid) {
    try {
        Scanner getInput = new Scanner(System.in);
        System.out.println("Enter dividend: ");
        int dividend = getInput.nextInt();
        System.out.println("Enter divisor: ");
        int divisor = getInput.nextInt();
        double result = dividend / divisor;
        valid = true;
    } catch (ArithmeticException e) {
        System.out.println("Input not valid.");
    }
}
```

```
Enter dividend:
10
Enter divisor:
0
Input not valid.
Enter dividend:
10
Enter divisor:
5
```

Range/data type validation with try/catch

Import: `java.util.InputMismatchException;`

We don't know how many times the user will input an incorrect value, so we add a while loop.

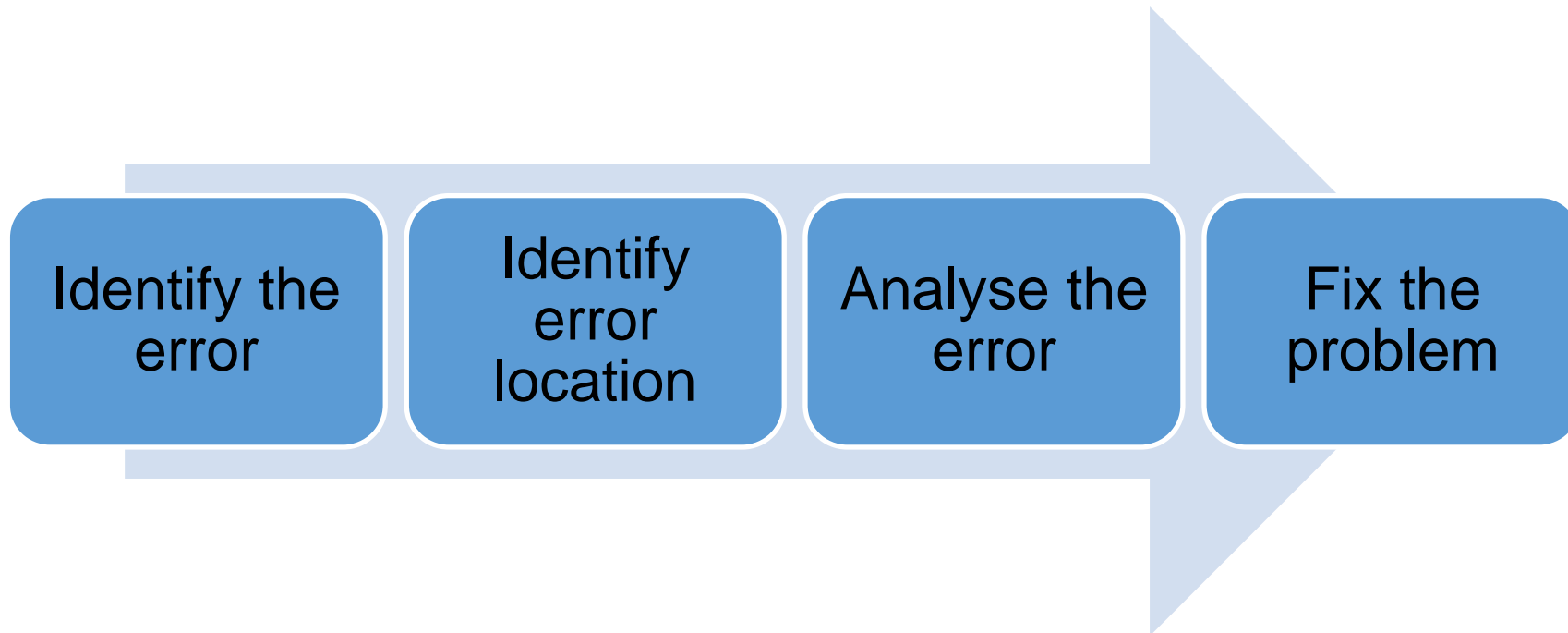
```
boolean correct = false;
while (!correct) {
    try {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter your age: ");
        int age = input.nextInt();
        if (age < 0 || age > 150){
            System.out.println("Age must be between 0 and 150.");
        }
        else correct = true;
    } catch (InputMismatchException e) {
        System.out.println("Age must be a number.");
    }
}
```

Good practices

- Use descriptive and informative variable names.
- Always declare the type of variable that is needed and not something else.
 - Example: Do not declare a variable as double if you need only an integer (saves memory).
- Always add comments explaining your code.
- Always indent the code.
- Make the program easy to understand by the user.

Debugging

- Debugging is the process of identifying and fixing errors or bugs in the code of a program.
- When the program does not work as expected, we need to study the code to find why any errors occurred.
- Improves understanding of the program code.



Debugging strategies

- Code tracing: read the code line by line
- Use print statements
- Use log files (with print statements)
- Explain the code to others
- Use a debugger at run-time

Rubber Duck Debugging

Rubber duck is a method for debugging

Steps:

- 1.- Take your rubber duck
- 2.- Explain to the duck your code, line-by-line. The more detailed your explanations, the easier will be to find your problem.
- 3.- Use the debugger to see the value of the variables at each line.
- 4.- Did you find the problem? If not, explain your code to the duck again, until you find the problem.

More info here:

https://en.wikipedia.org/wiki/Rubber_duck_debugging

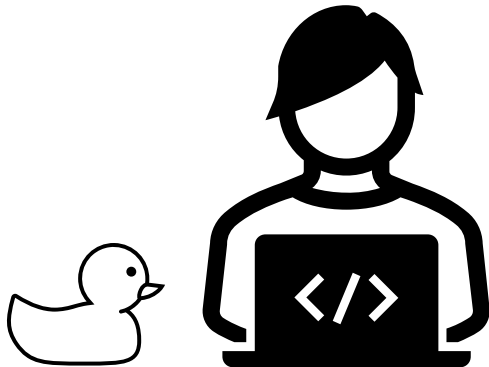


Rubber Duck Debugging: Example

This is your code, you want to print the first student ID (w123456) but instead it prints: w1234568.
You cannot find the problem.

```
String[] students_ID = {"w1234567", "w1234568", "w1234569"};  
System.out.println(students_ID[1]);
```

First I create a list of string ID's. I have 3 ID's.
Then I print the ID. To print the ID I get the ID in position 1.
In position 1 says that we have the value w1234568.
...
Aha! In Java the first element is in position 0, not 1.
My problem is solved. Thanks duck.



Independent Study

- In Learning Resources > Week 3 you will find a formative test to get feedback on the content of this lecture. Complete the test.
- Refer self study videos in Blackboard.
- Read Java for Everyone Chapter 2-3.
- Complete Tutorial 02 with HackerRank Questions.
- Submit Tutorial 01 answers to Blackboard (Deadline and submission links are provided in Blackboard)