Lecture 5

- Understand concepts of loops
- Different types of loops
  - while loop
  - do-while loop
  - for loop

Motivation

- Very often a program would repeat the same set of procedures several times
  - To compute the grades for different students
  - To move the car continuously
  - To create a moving sequence of images
- Loops allow a block of code to be executed repeated

Introduction to Loops

- Three types of loops
  - while loop
  - do-while loop
  - for loop
- Nested Loops
  - Similar to if statements, loops can also be nested

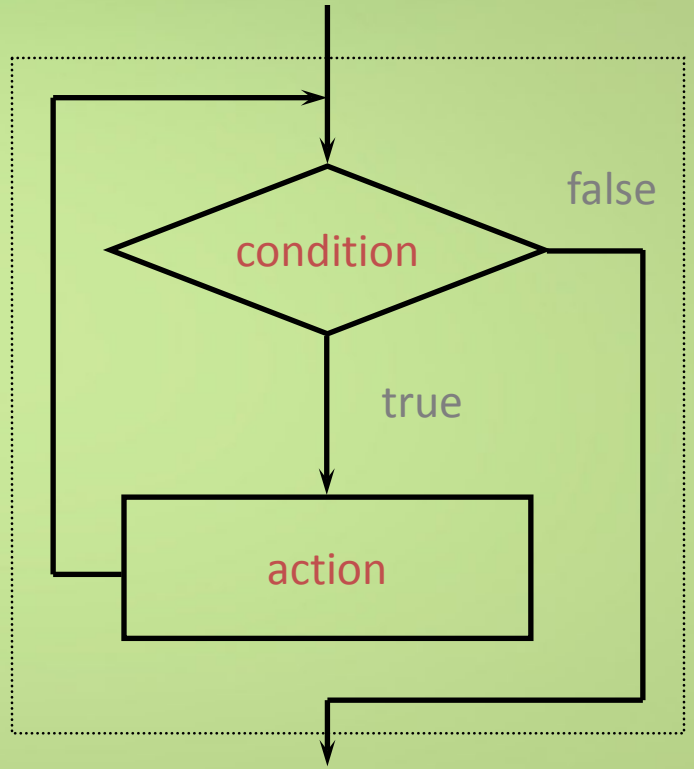
while Loop

- Syntax

  while (condition) {

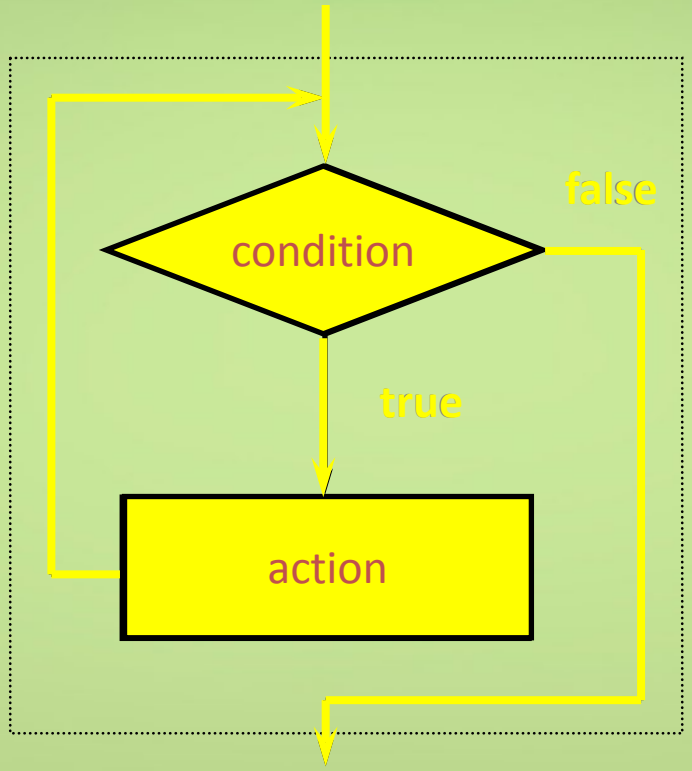
  // action ;

  }
- How does it work?
  - if condition is true then execute action
  - repeat action until condition becomes false
- Action can be a group of statements or a single statement.



A Loop Statement

```
while ( condition )
{
    action ;
}
```



Compute n!

- **n!** (n factorial) is defined as the product of all the integers from 0 to n:

  n! = 1*2*3*…*n (or n! = (n-1)!*n) and 0! = 1

- For example, 5! = 1*2*3*4*5 = 120
- Algorithm for computing n!
  - Initialize the intermediate result **t** to 1 (or 0!) and a **counter** to 1.
  - As long as **counter** is less than or equal n, repeat the computation of **t*counter** and increase **counter** by one.
    - 1! = 0!*1 = t*1 (set **t** to 1, which is 0!)
    - 2! = 1!*2 = **t***2 (update t to this new value, which is 2!)
    - 3! = 2!*3 = **t***3 (update t to this new value, which is 3!)
    - 4! = 3!*4 = **t***4 (update t to this new value, which is 4!)
    - 5! = 4!*5 = **t***5 (update t to this new value, which is 5!)

Compute n! (while loop)

Implement n! using a while loop

- Initialize the intermediate result **t** to 1 (or 0!) and a **counter** to 1
- As long as **counter** is less than or equal n, repeat the computation of **t*counter** and increase **counter** by one.

```
public static int factorial(int number) {
  int t = 1;     // initialize t to 1
  int counter = 1; // initialize counter to 1

  while(counter <= number){
    t = t * counter;
    counter = counter + 1;
  }
  return t;
}
```

Compute 2^n

- 2^n is 2 raised to the n-th power:

  $2^n = 2^{n-1} * 2$

- For example, $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$ …
- Algorithm for computing 2^n
  - Initialize the intermediate result **t** to 1 (or 2^0) and **counter** to 1.
  - As long as **counter** is less than or equal to n, repeat the computation of **t***2 and update **counter**.
    - $2^0 = 1 = t$ (set t to 1, which is 2^0)
    - $2^1 = 2^0 * 2 = t*2$ (update t to this new value, which is 2^1)
    - $2^2 = 2^1 * 2 = t*2$ (update t to this new value, which is 2^2)
    - $2^3 = 2^2 * 2 = t*2$ (update t to this new value, which is 2^3)
    - $2^4 = 2^3 * 2 = t*2$ (update t to this new value, which is 2^4)

Compute 2^n (while loop)

Implement 2^n using a while loop

- Initialize the intermediate result **t** to 1 (or 2^0) and **counter** to 1.
- As long as **counter** is less than or equal to n, repeat the computation of **t***2 and update **counter**.

```
public static int powerTwo(int number) {
  int t = 1;     // initialize t to 1
  int counter = 1; // initialize counter to 1

  while(counter <= number) {
    t = t * 2;
    counter = counter + 1;
  }
  return t;
}
```

How to compute m^n ?

Class or static method

- Class (or static) methods are declared using the static modifier.
  For example: `public static int factorial(int number)`
  `public static int powerTwo (int number)`
- Class methods can be invoked without the need for creating an instance of the class. They can be invoked outside the class with the class name:
  - ClassName.staticMethodName(parameters);
- Instance methods can access instance variable and methods as well as class variable and methods directly.
- Class methods can access class variables methods directly but **not** instance variables and instance methods – they must use an object reference.

Increment and Decrement Operators

- Java has special operators for incrementing (++) and decrementing (--) an integer by one.
- The ++ operator functions as follows:
  - ++a increments the value of a by one and the incremented value is used in the expression.
  - a++ uses the initial value of a in the expression and increments afterwards.

Increment and decrement operators

Operator	Name	Description
++a {y = ++a;}	Pre-increment	Increase **a** by 1 and then use the value of **a** in the assignment {a = a + 1; y = a;}
a++ {y = a++;}	Post-increment	Use the initial value of **a** in the assignment and then increase a by 1 {y = a; a = a + 1;}
--a {y = --a;}	Pre-decrement	Decrease **a** by 1 and then use the value of **a** in the assignment {a = a - 1; y = a;}
a-- {y = a--;}	Post-decrement	Use the initial value of **a** in the assignment then decrease **a** by 1 {y = a; a = a - 1;}

Example

```
//   Prefix and Postfix Increment operators
public void testPrePost ( )  {
      int a;
      int y;
      a = 4;
      IO.outputln("value of a:     " + a  );
      y = a++ + 5;
      IO.outputln ("value of y:    " + y );
      IO.outputln ("new value of a:  " +  a );

      a = 4;
      IO.outputln ("value of a:     " + a );
      y = ++a + 5;
      IO.outputln ("value of y:    " + y );
      IO.outputln ("new value of a:  " +  a  );
}
```

```
/*
Results:
value of a:         4
value of y:         9
new value of a:   5

value of a:         4
value of y:        10
new value of a:   5
*/
```

Shortcut Assignments

- Java has a set of shortcut operators for applying an operation to a variable and then assigning the result back to that variable.
- Shortcut assignments :

	shortcut	same as
*=	`a *= b;`	`a = a*b;`
/=	`a /= b;`	`a = a/b;`
+=	`a += b;`	`a = a+b;`
-=	`a -= b;`	`a = a-b;`
%=	`a %= b;`	`a = a%b;`

Shortcut Assignments

Examples

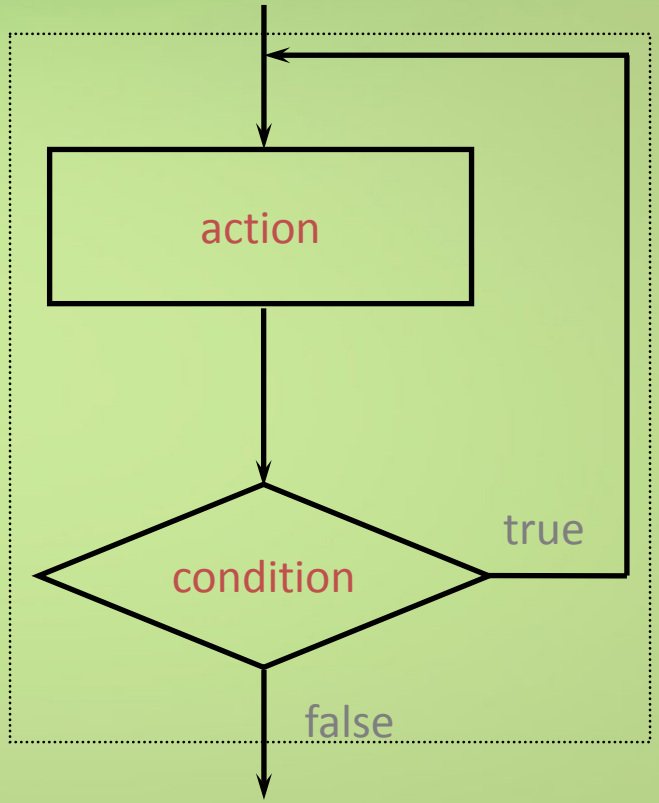
```
int i = 3;
i += 4;                          // i = i + 4
IO.outputln("i = " + i );  // i is now 7

double a = 3.2;
a *= 2.0;                        // a = a * 2.0
IO.outputln("a = " + a);   // a is now 6.4

int b = 15;
b %= 10;                         // b = b % 10
IO.outputln("b = " + b );  // b is now 5
```

do-while Loop

- **Syntax**

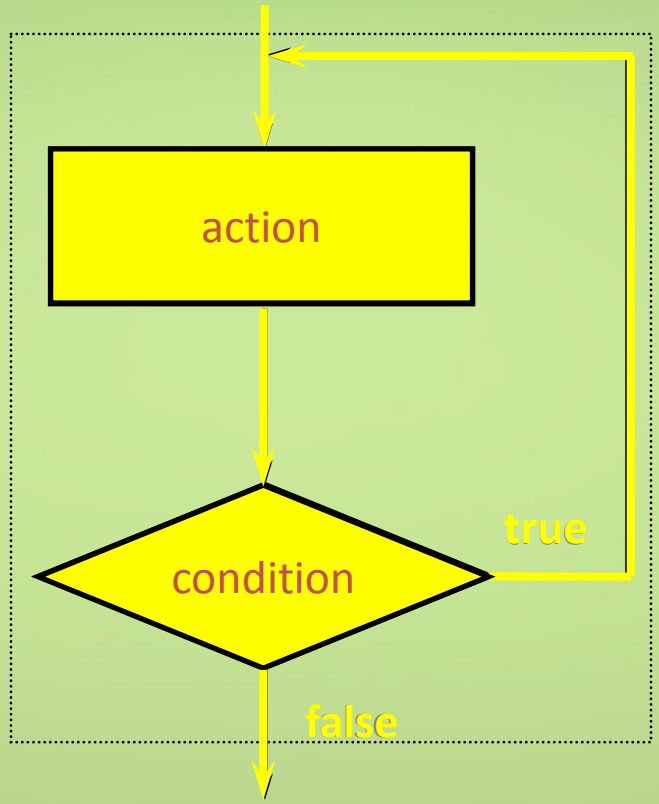
```
do {
    action;
} while (condition);
```

- **How does it work?**
  - execute action
  - if condition is true then execute action again
  - repeat this process until condition evaluates to false



do-while Loop

```
do {
    action;
} while (condition);
```



Compute n! (do-while loop)

Implement n! using a do-while loop

- Initialize the intermediate result **t** to 1 (or 0!) and a **counter** to 1
- Compute **t*counter** and increment **counter** by 1
- As long as **counter** is less than or equal to n, repeat the computation of **t*counter** and update **counter**

```
public static int factorial(int number) {
  int t = 1, counter = 1;

  do {
    t *= counter;  // t = t * counter
    counter += 1;  //counter = counter + 1
  } while(counter <= number); //don't forget the ';'
  return t;
}
```

Compute 2^n (do -while loop)

Implement 2^n using a do-while loop

- Initialize the intermediate result **t** to 1 (or 2^0) and **counter** to 1.
- Compute **t*2** and increment **counter** by 1
- As long as **counter** is less than or equal to n, repeat the computation of **t***2 and update **counter**.

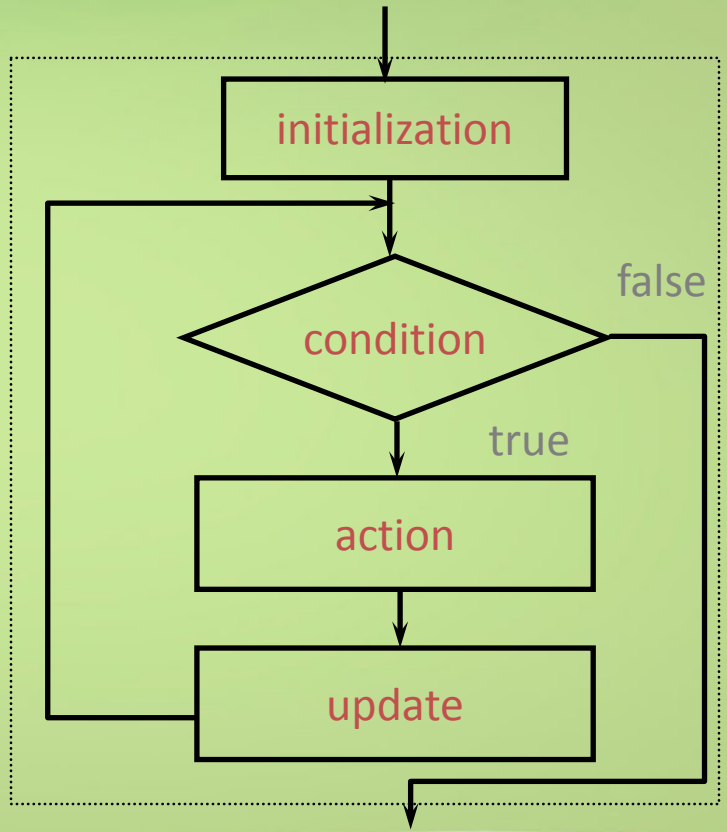
```
public static int powerTwo(int number) {
  int t = 1, counter = 1;

  do {
    t *= 2;                    // t = t*2
    counter += 1;              //counter = counter+ 1
  } while (counter <= number); //don't forget the ';'
  return t;
}
```

for Loop

- **Syntax:**

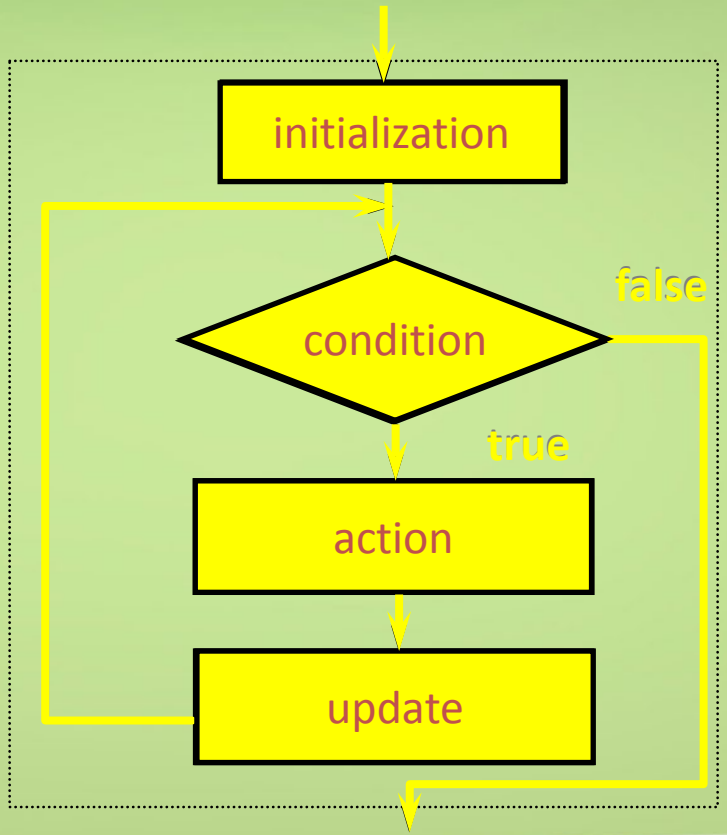
```
for (initialization;
     condition;
     update )
{
     action ;
}
```

- **How does it work?**
  - initialization
  - while condition is true, execute action *and* update



For Loop

```
for (initialization;
     condition;
     update )
{
    action ;
}
```



Compute n! (for loop)

Implement n! using a for loop

- Initialize the intermediate result **t** to 1 (or 0!) and **counter** is initialized in the for loop
- As long as **counter** is less than or equal to n, repeat the computation of **t*counter** and update **counter** using **counter++**

```
public static int factorial(int number) {
  int t = 1; int counter;

  //set up counter, condition check, update together
  for(counter=1; counter<=number; counter++) {
    t *= counter;
  }
  return t;
}
```

Compute n! (for loop)

Implement n! using a for loop

- Initialize the intermediate result **t** to 1 (or 0!) and **counter** is initialized in the for loop
- As long as **counter** is less than or equal to n, repeat the computation of **t*counter** and update **counter** using **counter++**

```
public static int factorial(int number) {
  int t = 1;

  //set up counter, condition check, update together
  for(int counter=1; counter<=number; counter++) {
    t *= counter;
  }
  return t;
}
```

Compute 2^n (for loop)

Implement 2^n using a for loop

- Initialize the intermediate result **t** to 1 (or 2^0) and **counter** is initialized in the for loop.
- As long as **counter** is less than or equal to n, repeat the computation of **t***2 and update **counter** using **counter++**

```
public static int powerTwo(int number) {
  int t = 1;

  //set up counter, condition check, update together
  for(int counter=1; counter<=number; counter++) {
    t *= 2;
  }
  return t;
}
```

Common Mistakes: Floating-point numbers

- Neither use **== (equal)** nor **!= (not equal)** on floating point numbers
- Reasons
  - Floating-point values are approximated
  - In this example, item !=0 may never be false
  - Infinite loop is resulted if the loop condition is always true

```
double item = 1;
double sum = 0;

while (item != 0) {
  sum = sum + item;
  item = item – 0.1;
}
```

Item starts with 1 and is reduced by 0.1 every time the loop body is executed

Common Loop Errors

```
while(balance != 0.0);
{
    balance = balance - amount;
}
// This will lead to an infinite loop!

for(int n=1; n<=count; n++);
{
    IO.outputln("hello");
}
// "hello" only printed once!
```

Which loop to use?

- Programmers are free to choose one of the three loops
- In general
  - **while loop**
    - The number of iterations is unknown (or unclear), and the loop body may not need to be executed
  - **do-while loop**
    - The number of iterations is unknown (or unclear), and the loop body is always executed at least once
  - **for loop**
    - The number of iterations is known (e.g. 100 times)

Example: Savings Account

Savings account:

- Bank account that earns interest from the account balance
- As an example, for an account with a principal of $1,000 that earns an annual interest of 10%, assuming that the interest is compounded annually.
- What would be the accumulated balance at the end of 5 years?

- 1st year: interest earned $1,000 * 10% or 100, new balance $1,100
- 2nd year: interest earned $1,100 * 10% or 110, new balance $1,210
- 3rd year: interest earned $1,210 * 10% or 121, new balance $1,331
- 4th year: interest earned $1,331 * 10% or 133.1, new balance $1,464.1
- 5th year: interest earned $1,464.1 * 10% or 146.41, new balance $1,610.51

Subclass and Inheritance

- A subclass is a class that is derived from another class (superclass).
  - public class SubclassName extends SuperClassName
- The class **Object** is the root of the Java class hierarchy.
- A subclass inherits all the fields and methods from its superclass.
- The keyword super can be used for a subclass to invoke the constructors or methods of its superclass.

An Example: Bank Account

```
import comp102x.IO;
/**
 * A bank account has a balance and an owner who can make
 * deposits to and withdrawals from the account.
 */
public class BankAccount {
   private double balance = 0.0;     // Initial balance is set to zero
   private String owner = "NoName";     // Name of owner

   /**
    *   Default constructor for a bank account with zero balance
    */
   public BankAccount ( ) { }
   /**
    *   Construct a balance account with a given initial balance and owner's name
    *   @param   initialBalance     the initial balance
    *   @param   name               name of owner
    */
   public BankAccount (double initialBalance, String name) {
      balance = initialBalance;
      owner = name;
   }
```

An Example: Bank Account

```
/**
 *  Method for depositing money to the bank account
 *  @param  dAmount    the amount to be deposited
 */
public void deposit(double dAmount) {
    balance = balance + dAmount;
}
/**
 *  Method for withdrawing money from the bank account
 *  @param  wAmount    the amount to be withdrawn
 */
public void withdraw(double wAmount) {
    balance = balance - wAmount;
}
/**
 *  Method for getting the current balance of the bank account
 *  @return  the current balance
 */
public double getBalance() {
    return balance;
}
```

An Example: Bank Account

```
import comp102x.IO;
/**
 * SavingsAccount is a subclass of BankAccount.
 */
public class SavingsAccount extends BankAccount {
    double interestRate;
    /**
     *  Constructor that makes use of the constructor from super class
     */
    public SavingsAccount (double initialBalance, String name, double rate)  {
        super(initialBalance, name);
        interestRate = rate;
    }
```

```
// -  A subclass inherits all the members including fields and
//    methods from its superclass.
// -  Constructors of the superclass are NOT inherited by subclasses
//     but can be invoked from the subclass.
```

An Example: Bank Account

```
/**
   *  compoundInterest computes the compound interest for a given duration
   *
   *  @param   duration        the number of times the interest is to be compounded
   */
public void compoundInterest (int duration)   {
    for (int i =1; i <= duration; i++) {
        double currentBalance = getBalance();
        deposit(currentBalance * interestRate);}
    }

public void setInterestRate(double rate) {
    interestRate = rate;
    }
}
```

Interest earned for the i^{th} period

```
// Formula for computing compound interest:
//      P ( 1 + r ) n
```