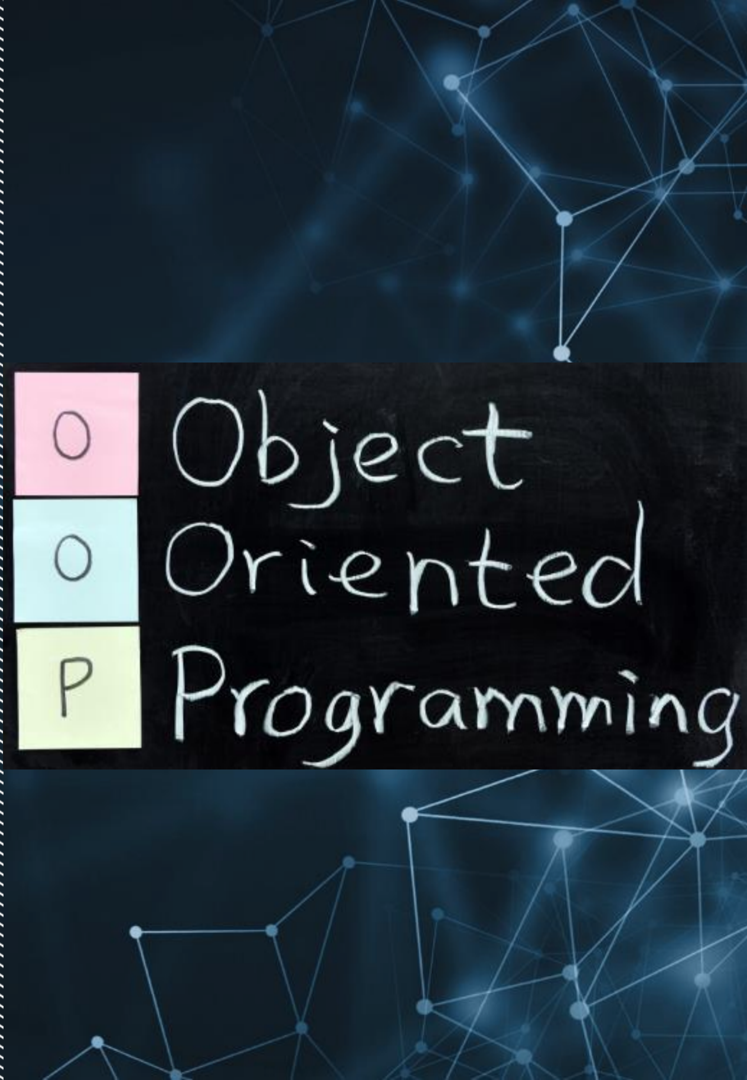
Object Oriented Programming



Agenda

- ✓Classes and Object
- ✓Inheritance
- ✓Polymorphism
- ✓Abstraction
- ✓Encapsulation
- ✓Abstract Class vs Interface

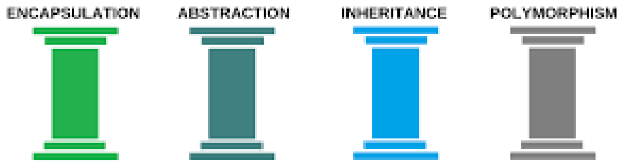
Object Oriented Programming

- *"OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things."*
  ~ Alan Kay

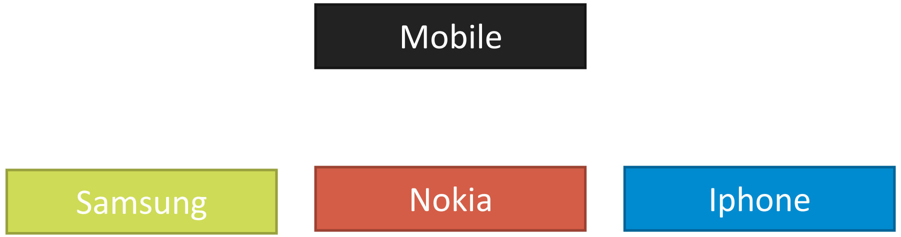
- Object Oriented Programming is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*).

- First Object Oriented programming languages- simula developed in 1967

- There are mainly four pillars (features) of OOP.
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism



Understanding Object Oriented Programming using Example - Java



- Mobile phones are invented with the basic features like Calling ,Receiving a call & Messaging

Object

- Any real-world entity which can have some characteristics, or which can perform tasks is called as Object.
- Object is also called an instance i.e. a copy of entity in programming language

What can be Object in the Mobile Phone Use case?

- A Mobile Manufacturing company can be an Object.

```
Mobile mobile = new Mobile ();   //Object creation
```

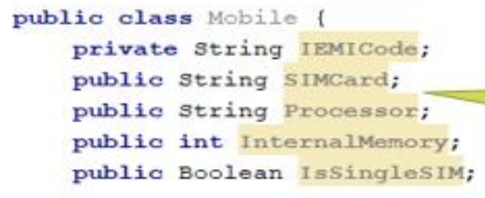
Objects Advantage

- Modularity : Object can be written and maintained independently .
- Information-hiding : Internal implementation remain hidden from the outside world .
- Code re-use : Use Pre-existing objects
- Pluggability and debugging ease : If a bolt breaks, you replace *it*, not the entire machine.

Class

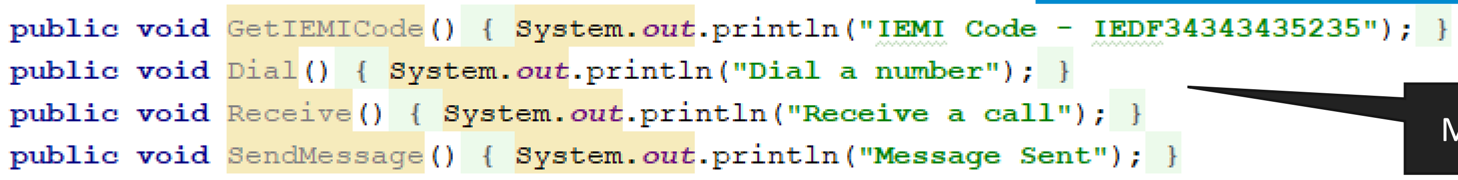
- A class in OOP is a plan which describes the object - Blueprint of Object

- Considering the above example the Mobile can be a class, which has some attributes like **Profile Type, IMEI Number, Processor,** and some more. It can have operations **like Dial, Receive and SendMessage.**



```
public class Mobile {
    private String IEMICode;
    public String SIMCard;
    public String Processor;
    public int InternalMemory;
    public Boolean IsSingleSIM;
```

Attributes



```
public void GetIEMICode() { System.out.println("IEMI Code - IEDF34343435235"); }
public void Dial() { System.out.println("Dial a number"); }
public void Receive() { System.out.println("Receive a call"); }
public void SendMessage() { System.out.println("Message Sent"); }
```

Methods/Functions

SOLID principle :

- SRP (The Single Responsibility Principle) - A class should have one, and only one responsibility
- OCP (The Open Closed Principle) - You should be able to extend a classes behavior, without modifying it. (Inheritance)
- LSP (The Liskov Substitution Principle) - Derived classes must be substitutable for their base classes. (Polymorphism)
- ISP (The Interface Segregation Principle) -Make fine chopped interface instead of huge interface as client cannot be forced to implement an interface which they don't use.
- DIP (The Dependency Inversion Principle) - Depend on abstractions, not on concretions. (Abstraction)

Class Specifiers

public	Class is available from anywhere
friendly	Class is available only within the package
abstract	An instance of the class can not be created, it is necessary to inherit from such a class and create instances of the derived classes
final	Class inheritance is forbidden

Method Specifiers

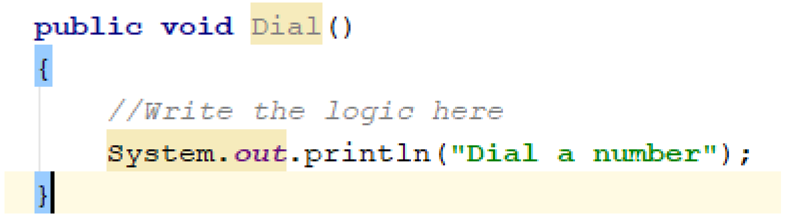
public	Available from anywhere
protected	Available from subclasses
private	Available only within the class
static	Available without creating an object
final	Overriding in inheritors is forbidden
abstract	Requires implementation in inheritors
friendly	Available only within the package
synchronized	Used when working with threads
strictfp	«Strict floating point» (on all platforms)
native	Implemented with Java Native Interface

Abstraction

- Abstraction allows us to expose limited data and functionality of objects publicly and hide the actual implementation. It is the most important pillar in OOPS

Abstract Features of Mobile:

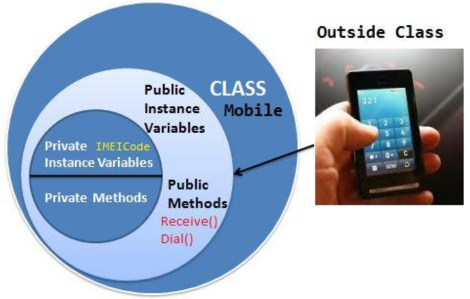
- Dialing a number will call dial method internally which concatenate the numbers and displays it on screen but what internally is happening is not exposed to outer world.
- Clicking on green button [call] actual send signals to calling person's mobile but we are unaware of how it is being done.

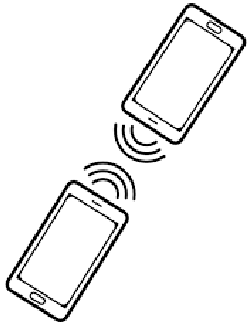


```
public void Dial()
{
    //Write the logic here
    System.out.println("Dial a number");
}
```

Encapsulation

- Encapsulation is defined as the process of enclosing one or more details from outside world through access right
- Both Abstraction & Encapsulation works hand in hand
- Encapsulation provides the level of access right to that visible details.
  i.e. – It implements the desired level of abstraction.
- Abstraction says what details to be made visible





Example:

Talking about Bluetooth which we usually have it in our mobile. When we switch on a Bluetooth, I can connect to another mobile or Bluetooth enabled devices but I'm not able to access the other mobile features like dialing a number, accessing inbox etc. This is because, Bluetooth feature is given some level of abstraction.

Another point is when mobile A is connected to mobile B via Bluetooth whereas mobile B is already connected to mobile C then A is not allowed to connect C via B. This is because of accessibility restriction.

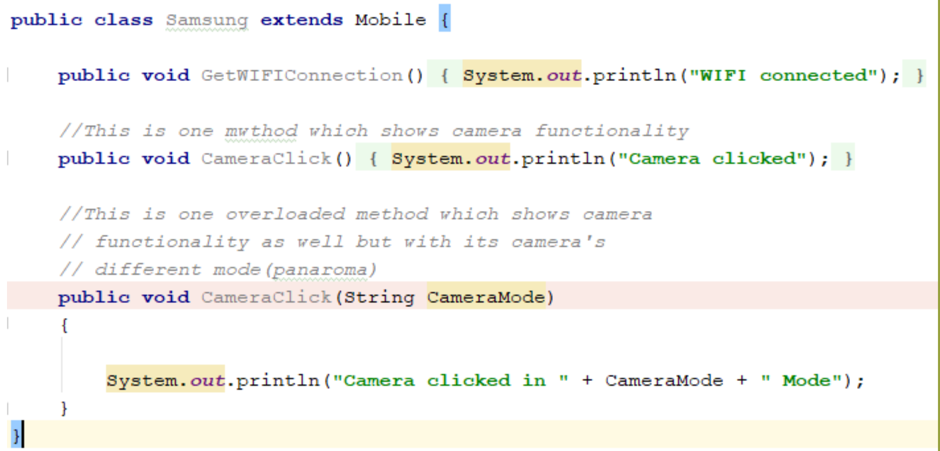
Accessibility: `private String IEMICode;`

Access Modifiers : public, private, default

Polymorphism

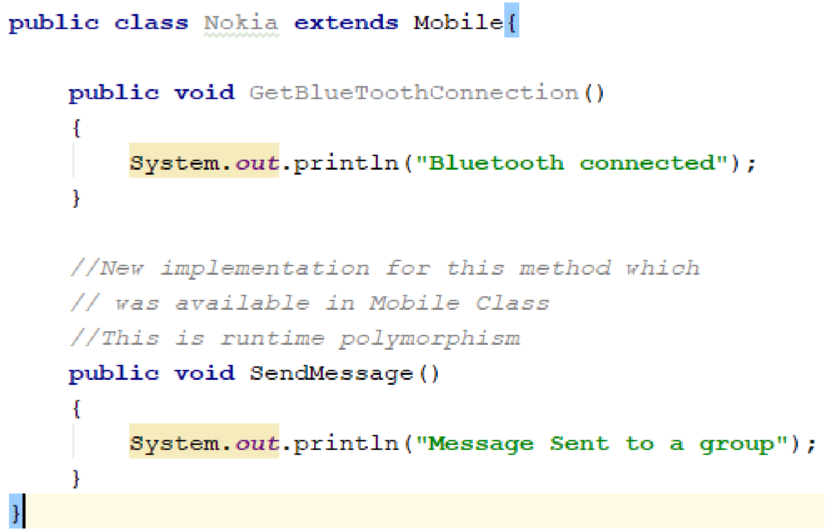
Polymorphism - many forms of single entity

Static polymorphism



```java
public class Samsung extends Mobile {

    public void GetWIFIConnection() { System.out.println("WIFI connected"); }

    //This is one mwthod which shows camera functionality
    public void CameraClick() { System.out.println("Camera clicked"); }

    //This is one overloaded method which shows camera
    // functionality as well but with its camera's
    // different mode(panaroma)
    public void CameraClick(String CameraMode)
    {

        System.out.println("Camera clicked in " + CameraMode + " Mode");
    }
}
```

Dynamic polymorphism



```java
public class Nokia extends Mobile{

    public void GetBlueToothConnection()
    {
        System.out.println("Bluetooth connected");
    }

    //New implementation for this method which
    // was available in Mobile Class
    //This is runtime polymorphism
    public void SendMessage()
    {
        System.out.println("Message Sent to a group");
    }
}
```

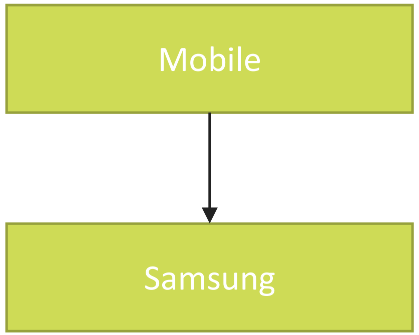
Inheritance

Inheritance is a mechanism in which one object acquires all the properties and behaviors of a parent object

- Single level Inheritance
- Multi level Inheritance
- Hierarchical level Inheritance
- Hybrid Inheritance
- Multiple Inheritance

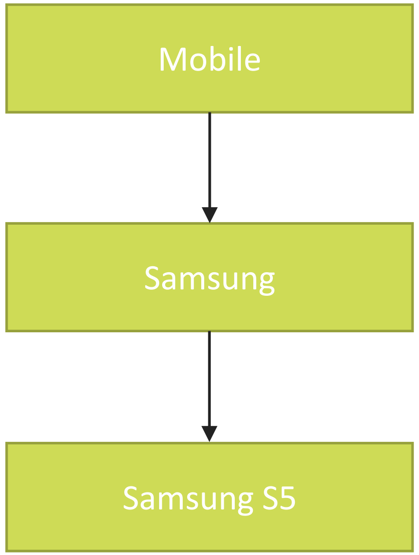
Single Level Inheritance

In Single level inheritance, there is single base class & a single derived class i.e. - A base mobile features is extended by Samsung brand.



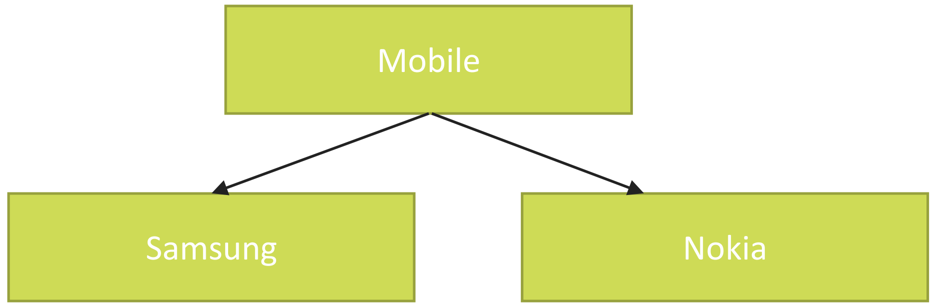
Multi Level Inheritance

In Multilevel inheritance, there is more than one single level of derivation. i.e. - After base features are extended by Samsung brand. Now Samsung brand has manufactured its new model with new added features or advanced OS like Android OS, v10.0 (**Q**). From generalization, getting into more specification.



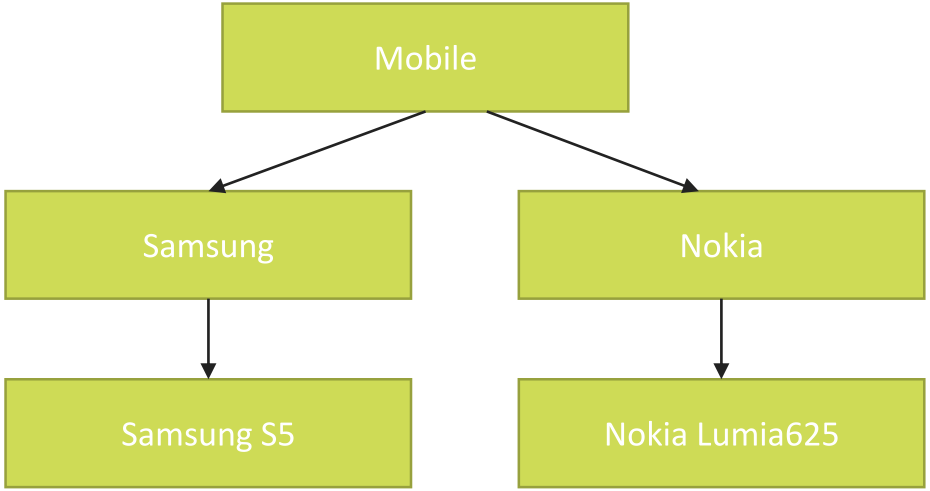
Hierarchical Inheritance

In this type of inheritance, multiple derived class would be extended from base class, it's similar to single level inheritance but this time along with Samsung, Nokia is also taking part in inheritance.



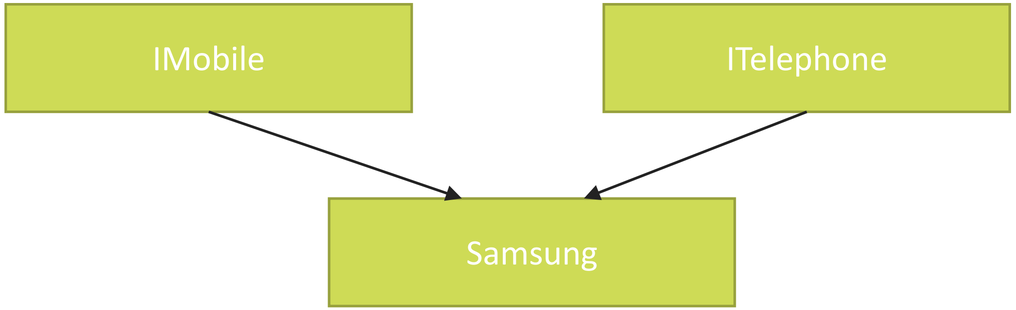
Hybrid Inheritance

Single, Multilevel, & hierarchal inheritance all together construct a hybrid inheritance.



Interface- Multiple Inheritance

Single, Multilevel, & hierarchal inheritance all together construct a hybrid inheritance.



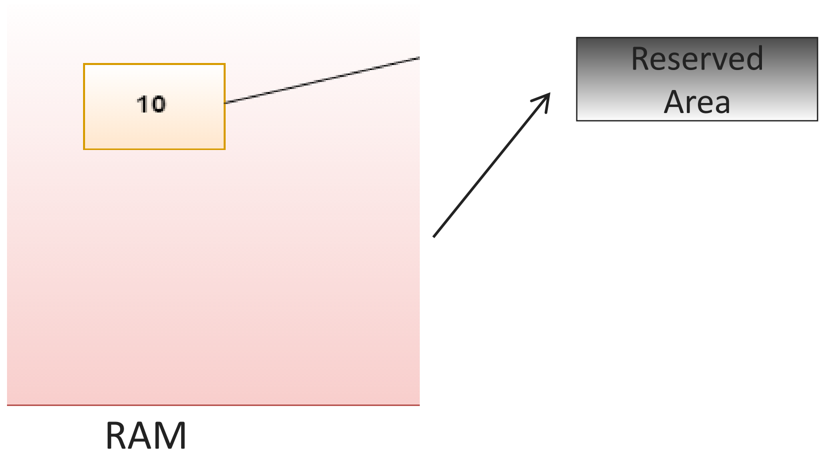
Basic Specifiers

Specifier	Class	Package	Inheritors	Anywhere
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default *(without modifier)*	Yes	Yes	No	No
private	Yes	No	No	No

Java Variables

Java Variables

Variable is name of *reserved area allocated in memory*. In other words, it is a *name of memory location*. It is a combination of "vary + able" that means its value can be changed.



Types of Variables

Local Variable

Instance Variable

Static Variable

Local Variable

A variable declared inside the body of the method is called local variable.

```
class A{
    void method(){
        int n=90; //Local variable
    }
}//end of class
```

Types of Variables

Static variable:

A variable declared inside the class but outside the body of the method, is called instance variable.

```
class A{
          static int data=50;//static variable
     void method()
         {
         int n=90;//local variable
         }
     }//end of class
```

Types of Variables

Instance variable:

A variable declared inside the class but outside the body of the method, is called instance variable.

```
class A{
    int data=50;//instance variable
            void method()
            {
               int n=90;//local variable
            }
          }//end of class
```

Java Data Types

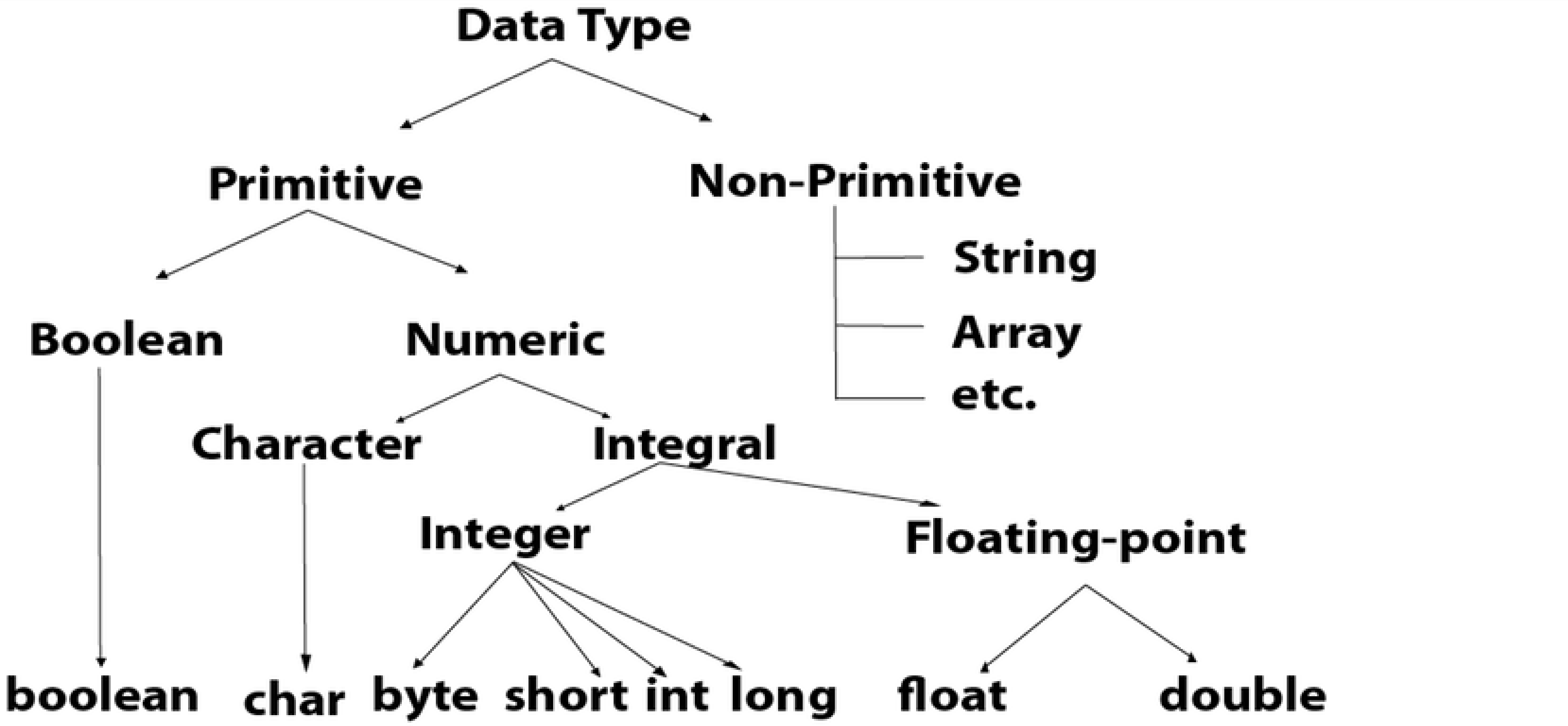
Data Types

There are two types of data types in Java

Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.

Data Types


Data Type
Primitive
Non-Primitive
String
Array
etc.
Boolean
Numeric
Character
Integral
Integer
Floating-point
boolean
char
byte
short
int
long
float
double

Data Types

Boolean Data Type

The Boolean data type is used to store only two possible values: true and false.

Example: Boolean one = false

Int Data Type

The int data type is generally used as a default data type for integral values.

Example: int a = 100000

Long Data Type

The long data type is a 64-bit two's complement integer.

Example: long a = 100000L

Data Types

Double Data Type

The double data type is a double-precision 64-bit IEEE 754 floating point.

Example: double d1 = 12.3

Char Data Type

The char data type is used to store characters.

Example: char letterA = 'A'

Control Statements

Java If-else Statement

The Java *if statement* is used to test the condition. It checks Boolean condition: *true* or *false*.

if statement

if-else statement

if-else-if ladder

nested if statement

Java if Statement

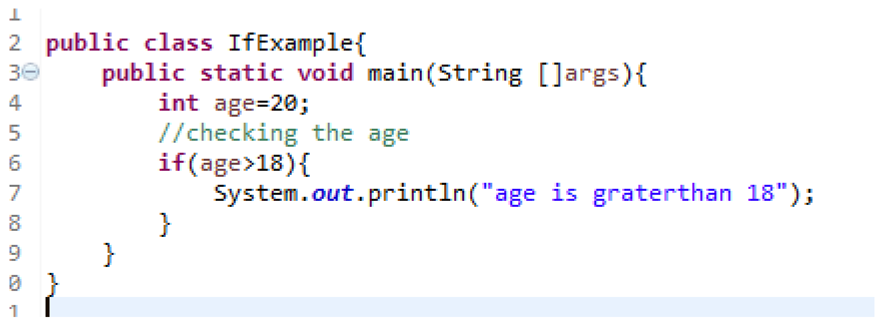
The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax: **if**(condition){

//code to be executed

}

Example:



```
public class IfExample{
    public static void main(String []args){
        int age=20;
        //checking the age
        if(age>18){
            System.out.println("age is graterthan 18");
        }
    }
}
```

Java If-else Statement

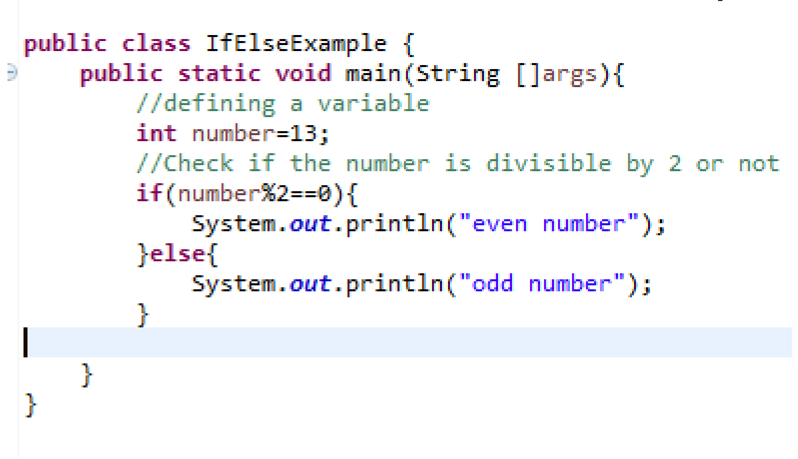
Java if-else Statement:

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
                  if(condition){
         //code if condition is true
        }else{
                  //code if condition is false
                  }
```

Example:



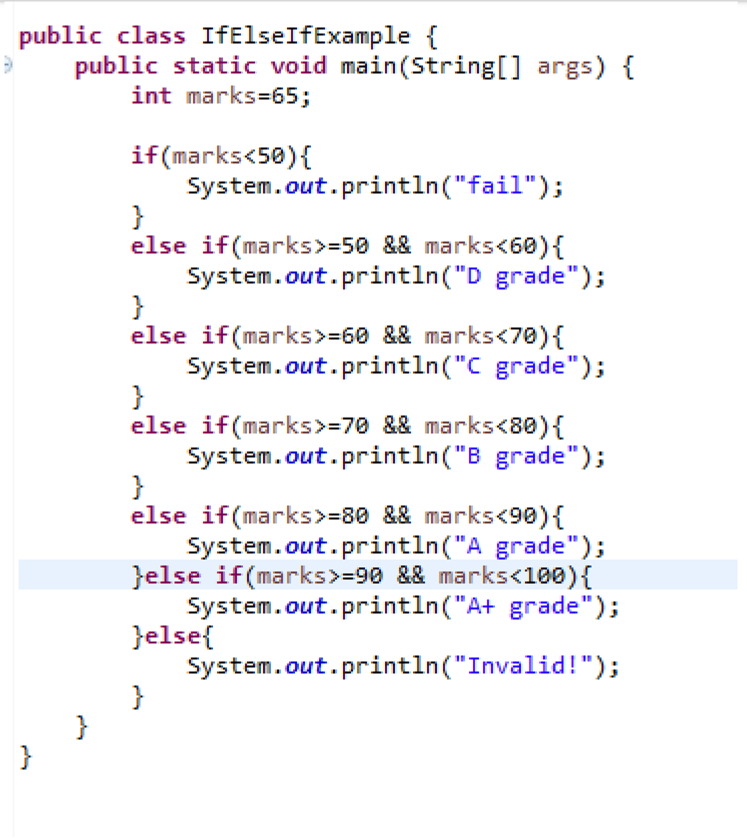
```
public class IfElseExample {
    public static void main(String []args){
        //defining a variable
        int number=13;
        //Check if the number is divisible by 2 or not
        if(number%2==0){
            System.out.println("even number");
        }else{
            System.out.println("odd number");
        }

    }
}
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

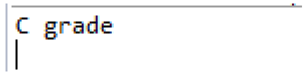
Syntax:

```
if(condition1){
          //code to be executed if condition1 is true
}else if(condition2){
  //code to be executed if condition2 is true
}  else if(condition3){
                  //code to be executed if condition3 is true
                 }  ...
                else{

//code to be executed if all the conditions are false
}
```



```
public class IfElseIfExample {
    public static void main(String[] args) {
        int marks=65;

        if(marks<50){
            System.out.println("fail");
        }
        else if(marks>=50 && marks<60){
            System.out.println("D grade");
        }
        else if(marks>=60 && marks<70){
            System.out.println("C grade");
        }
        else if(marks>=70 && marks<80){
            System.out.println("B grade");
        }
        else if(marks>=80 && marks<90){
            System.out.println("A grade");
        }else if(marks>=90 && marks<100){
            System.out.println("A+ grade");
        }else{
            System.out.println("Invalid!");
        }
    }
}
```

Output:



```
C grade
```

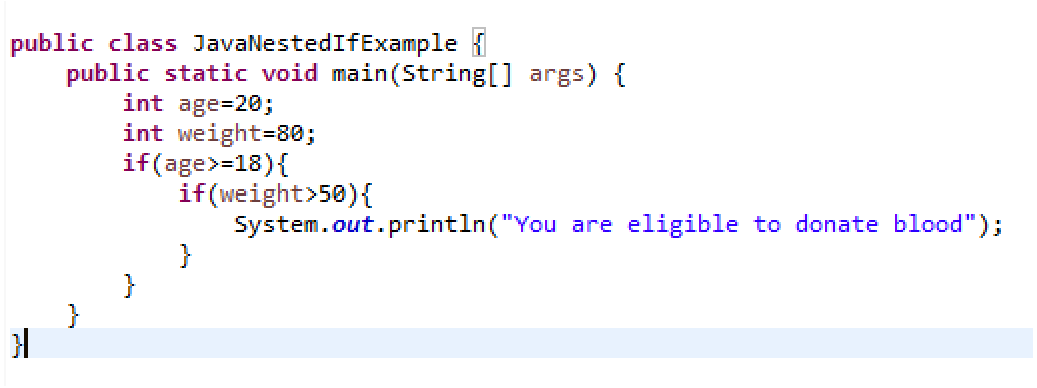
Java Nested if statement

The nested if statement represents the *if block within another if block*.

Syntax:

```
if(condition){
        //code to be executed
    if(condition){
  //code to be executed
}
}
```

Example:



```
public class JavaNestedIfExample {
    public static void main(String[] args) {
        int age=20;
        int weight=80;
        if(age>=18){
            if(weight>50){
                System.out.println("You are eligible to donate blood");
            }
        }
    }
}
```

Example:

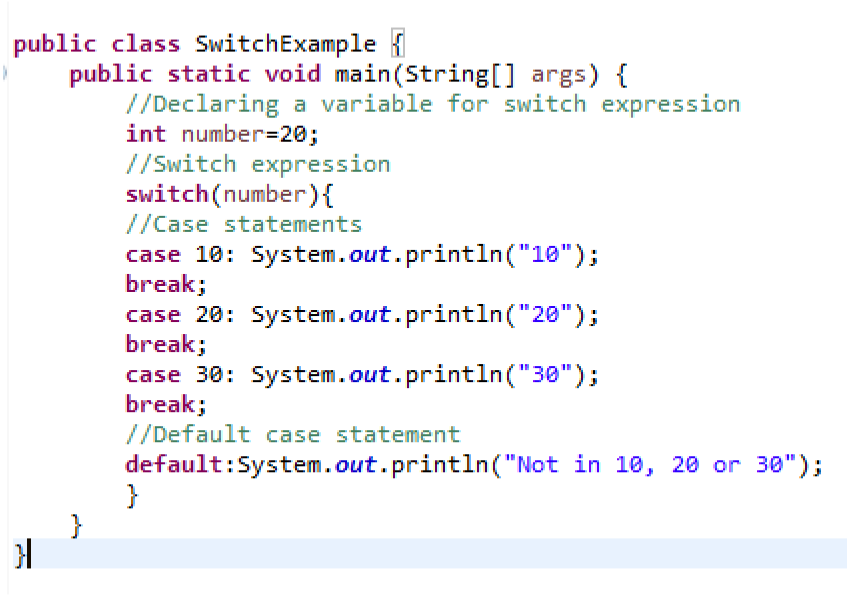
```
You are eligible to donate blood
```

Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```



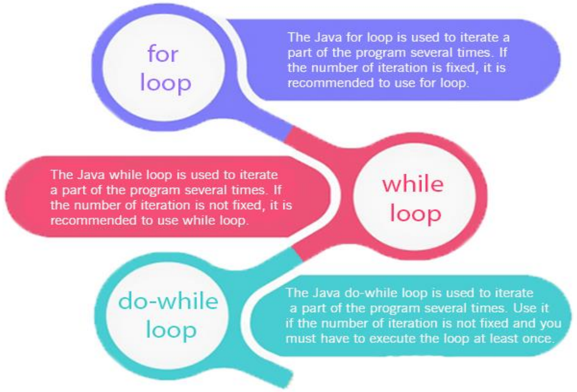
```
public class SwitchExample {
    public static void main(String[] args) {
        //Declaring a variable for switch expression
        int number=20;
        //Switch expression
        switch(number){
        //Case statements
        case 10: System.out.println("10");
        break;
        case 20: System.out.println("20");
        break;
        case 30: System.out.println("30");
        break;
        //Default case statement
        default:System.out.println("Not in 10, 20 or 30");
        }
    }
}
```

Output: 20

Loops

There are three types of loops in java.

- for loop
- while loop
- do-while loop

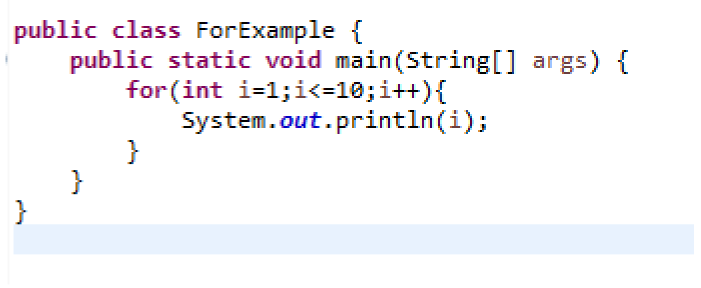

for loop
The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
while loop
The Java while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.
do-while loop
The Java do-while loop is used to iterate a part of the program several times. Use it if the number of iteration is not fixed and you must have to execute the loop at least once.

For Loop

Syntax:

```
for(initialization;condition;incr/decr){
  //statement or code to be executed
}
```

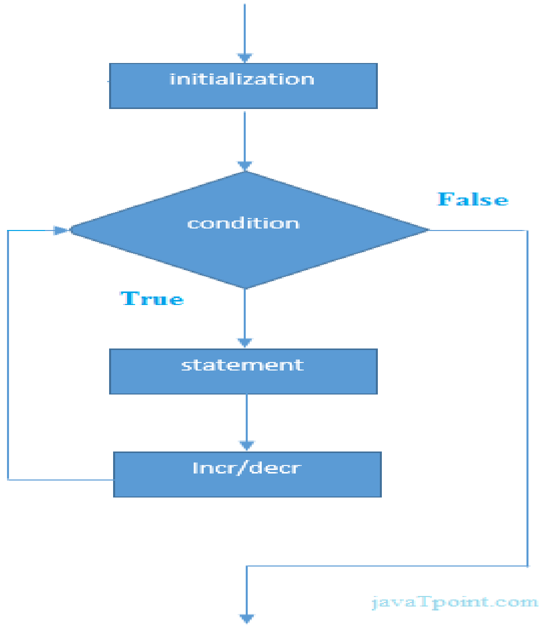
Example:



```
public class ForExample {
    public static void main(String[] args) {
        for(int i=1;i<=10;i++){
            System.out.println(i);
        }
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```



While Loop

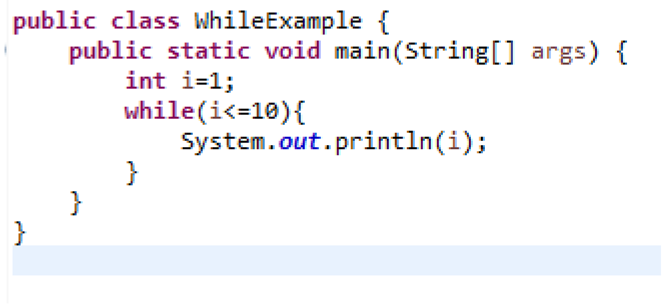
- The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax: **while**(condition){

//code to be executed

}

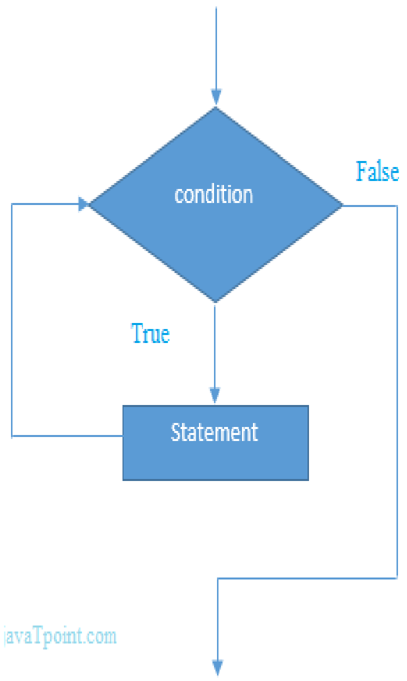
Example:



```
public class WhileExample {
    public static void main(String[] args) {
        int i=1;
        while(i<=10){
            System.out.println(i);
        }
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```



Do While Loop

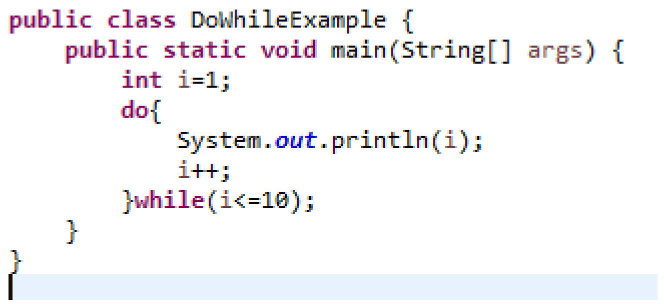
- The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax: **do**{

//code to be executed

}**while**(condition);

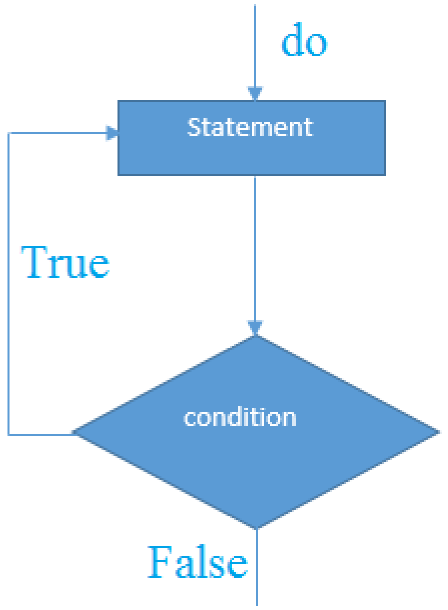
Example:



```
public class DoWhileExample {
    public static void main(String[] args) {
        int i=1;
        do{
            System.out.println(i);
            i++;
        }while(i<=10);
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```



Break

- The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

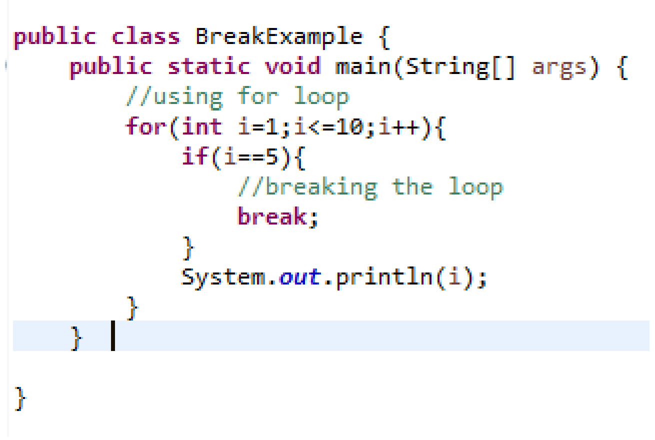
We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

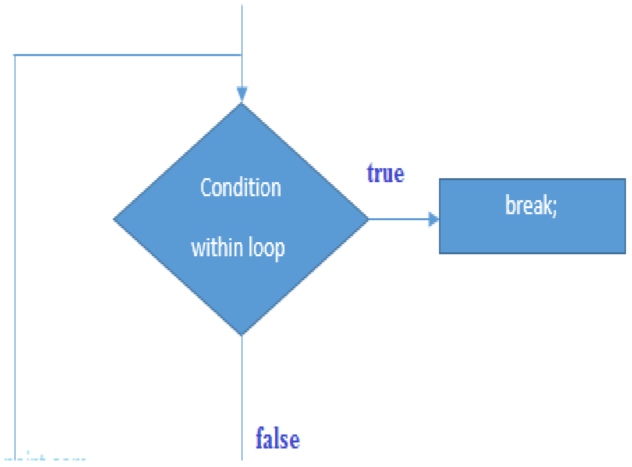
jump-statement;

break;

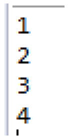
Example:



```
public class BreakExample {
    public static void main(String[] args) {
        //using for loop
        for(int i=1;i<=10;i++){
            if(i==5){
                //breaking the loop
                break;
            }
            System.out.println(i);
        }
    }

}
```



Output:



```
1
2
3
4
```

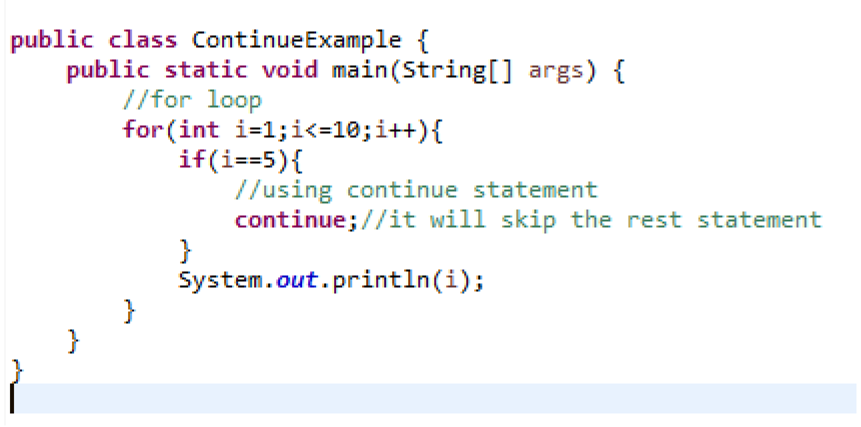
Continue

The Java *continue statement* is used to continue the loop.

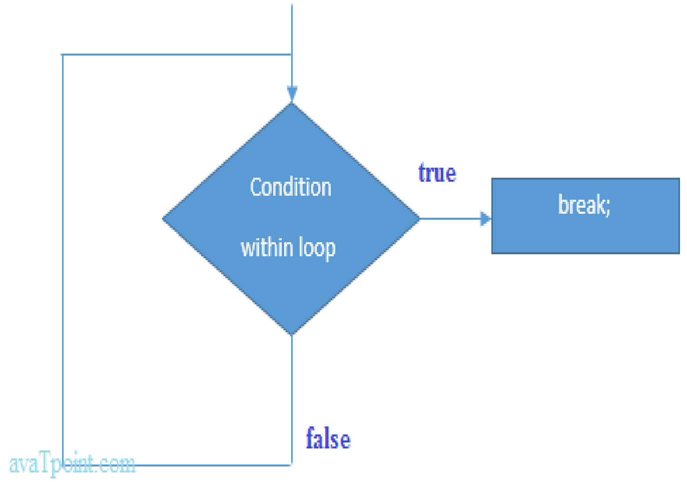
Syntax:

jump-statement;
continue;

Example:



```
public class ContinueExample {
    public static void main(String[] args) {
        //for loop
        for(int i=1;i<=10;i++){
            if(i==5){
                //using continue statement
                continue;//it will skip the rest statement
            }
            System.out.println(i);
        }
    }
}
```



Output:

```
1
2
3
4
6
7
8
9
10
```

STRING

String

- Reference: http://docs.oracle.com/javase/8/docs/api/java/lang/String.html
- No need to use constructor every time.
- Immutable.
- Concatenation with `+`
- Equal evaluation : `myString.equals(otherString)`
- `public static String.format(String format, Object... args)`
- `StringBuilder` and `StringBuffer` to manipulate String
- Important method: `Object.toString()`

String

CREATION

```
String str = "Hi!";
```

```
Char data[]= {‘H’,’I’};
String str = new String(data);
```

BEWARE OF NULL

```
String nullString = null;
if (nullString != null) {
// Check
     ...
}
 nullString.isEmpty(); //
NPE!
```

MANY USEFUL FEATURES

```
   int length = s.length();
   char firstChar = s.charAt(0);
   char[] carr = s.toCharArray();

   boolean isExecutable = s.endsWith(".exe");
   boolean isEmpty = s.isEmpty();
   int aidx = s.indexOf('a');
   boolean hasEM = s.contains('!');

   String upper = s.toUpperCase();
   String hai = s.substring(1, 3);
   String haiThere = hai + " there".
   String[] haiThereArr = haiThere.split(" ");
```

String

- Iteration of its chars:

```
for (int i=0; i<s.length(); ++i) {
      char act = s.charAt(i);
      ...
  }

  for (char act : s.toCharArray()) {
      ...
  }
```

- `replaceAll()`, `split()`: use with regexps, see Pattern at

http://download.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

- Comparison: `equals()`

```
boolean b1 = "a" == "a";        // Might be true
boolean b2 = "a".equals("a"); // Expected behaviour
```

- Don't forget: immutable structure!

```
String string = "AAAxAAA";
string.replace('x', 'A');
System.out.println(string); // "AAAxAAA"
string = string.replace('x', 'A');
System.out.println(string); // "AAAAAAA"
```

CharSeq's

- Use StringBuilder/StringBuffer:

```
StringBuffer sb = new StringBuffer();
sb.append("Hello ").append("World");
sb.reverse();
System.out.println( sb.toString() ); // "dlroW olleH"
sb.reverse();
sb.setCharAt(6, '-');
System.out.println( sb.toString() ); // "Hello-World"
sb.deleteCharAt(6);
System.out.println( sb.toString() ); // "HelloWorld"
sb.delete(0, sb.length() );
System.out.println( sb.toString() ); // ""
```

- StringBuffer is thread-safe (since 1.0)
- StringBuilder is not (since 1.5)

Home Task

- OOP's

Thank you!