OBJECT ORIENTED PROGRAMMING USING JAVA

OUTLINE

- Variables in java
- Methods

TYPES OF VARIABLES

- There are three types of variables in Java:
  1. Local Variables
  2. Instance Variables
  3. Static Variable

LOCAL VARIABLES

- A variable defined within a block or method or constructor is called local variable.
- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
- **Initialisation of Local Variable is Mandatory.**

LOCAL VARIABLES

```
public class Main
{
    public static void main(String[] args) {
        int a;
        System.out.println(a);
    }
}
```

```
public class Main
{
    public static void main(String[] args) {
        int a=10;
        System.out.println(a);
    }
}
```

LOCAL VARIABLES

```java
public class Main
{
    public static void main(String[] args) {
        int a;
        System.out.println(a);
    }
}
```

```
Main.java:13: error: variable a might not have been initialized
                System.out.println(a);
```

```java
public class Main
{
    public static void main(String[] args) {
        int a=10;
        System.out.println(a);
    }
}
```

```
10
```

LOCAL VARIABLES

```
public class Main
{
    public static void main(String[] args) {
        if (true)
        {
        int a=10;
        }
        System.out.println(a);
    }
}
```

LOCAL VARIABLES

```java
public class Main
{
    public static void main(String[] args) {
        if (true)
        {
        int a=10;
        }
        System.out.println(a);
    }
}
```

```
Main.java:16: error: cannot find symbol
                    System.out.println(a);
```

INSTANCE VARIABLES

- Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- Initialisation of Instance Variable is not Mandatory. **Its default value is 0**
- Instance Variable can be accessed only by creating objects.

INSTANCE VARIABLES

```
public class Main
{
    int num;
    int num1=10;
    float num2;
    public static void main(String[] args) {

        System.out.println(num+ " "+num1 + ""+ num2  );
    }
}
```

INSTANCE VARIABLES

```java
public class Main
{
    int num;
    int num1=10;
    float num2;
    public static void main(String[] args) {

        System.out.println(num+ " "+num1 + ""+ num2  );
    }
}
```

```
Main.java:16: error: non-static variable num cannot be referenced from a static context
            System.out.println(num+ " "+num1 + ""+ num2  );
                               ^
```

INSTANCE VARIABLES

```
public class Main
{
    int num;
    int num1=10;
    float num2;
    public static void main(String[] args) {
        Main obj=new Main();

        System.out.println(obj.num+ " "+obj.num1 + " "+ obj.num2  );
    }
}
```

INSTANCE VARIABLES

```
public class Main
{
    int num;
    int num1=10;
    float num2;
    public static void main(String[] args) {
        Main obj=new Main();

        System.out.println(obj.num+ " "+obj.num1 + " "+ obj.num2  );
    }
}
```

STATIC VARIABLES

- Static variables are also known as Class variables.
- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- **Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.**
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- **Initialisation of Static Variable is not Mandatory. Its default value is 0**
- If we access the static variable like Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name to class name automatically.
- If we access the static variable without the class name, Compiler will automatically append the class name.

STATIC VARIABLES

```
public class Main
{
    static int num;
    static int num1=10;

     public static void main(String[] args) {

         System.out.println(num+ " "+num1);
     }
}
```

STATIC VARIABLES

```
public class Main
{
    static int num;
    static int num1=10;

     public static void main(String[] args) {

         System.out.println(num+ " "+num1);
     }
}
```

```
0 10
```

METHODS

- We have used some existing methods without fully understanding their implementation
- System.out’s **print, println**
- String's **length, charAt, indexOf, toUpperCase**
- Scanner's nextDouble, nextInt
- BankAccount's withdraw, deposit
- Java has thousands of methods
- We often need to create our own

NEED OF METHODS

```
public class Main
{
        int a=10;
        int b=20;
        System.out.println(a+b);

}
```

NEED OF METHODS

```
public class Main
{
        int a=10;
        int b=20;
        System.out.println(a+b);

}
```

Compile time Error

Therefore in JAVA direct business logics are not allowed

NEED OF METHODS

Solution:

```
public class Main
{
        int a=10;
        int b=20;
        void add()
        {
        System.out.println(a+b);
        }
}
```

NEED OF METHODS

Solution:

```
public class Main
{
        int a=10;
        int b=20;
        void add()
        {
        System.out.println(a+b);
        }
}
```

But here we will get runtime error: Main method not found in class Main

METHODS

- Methods are used to write the logics of the application.
- A program that provides some functionality can be long and contains many statements
- A method groups a sequence of statements and should provide a well-defined, easy-to-understand functionality.

- Two types of methods in java:
  - **Instance method** (linked with object creation and can be accessed by only object)
  - **Static method.** (linked with class loading and can be accessed by only class)

WHAT METHOD HEADINGS TELL US

- Method headings provide the information needed to use it they show us how to send messages

```
public String substring(int beginIndex, int endIndex)
   1       2           3      5        4          5       4
```

- 1 Where is the method accessible
- 2 What does the method evaluate to?
- 3 What is the method name?
- 4 What type arguments are required?
- 5 How many arguments are required?

Syntax to declare a Method:

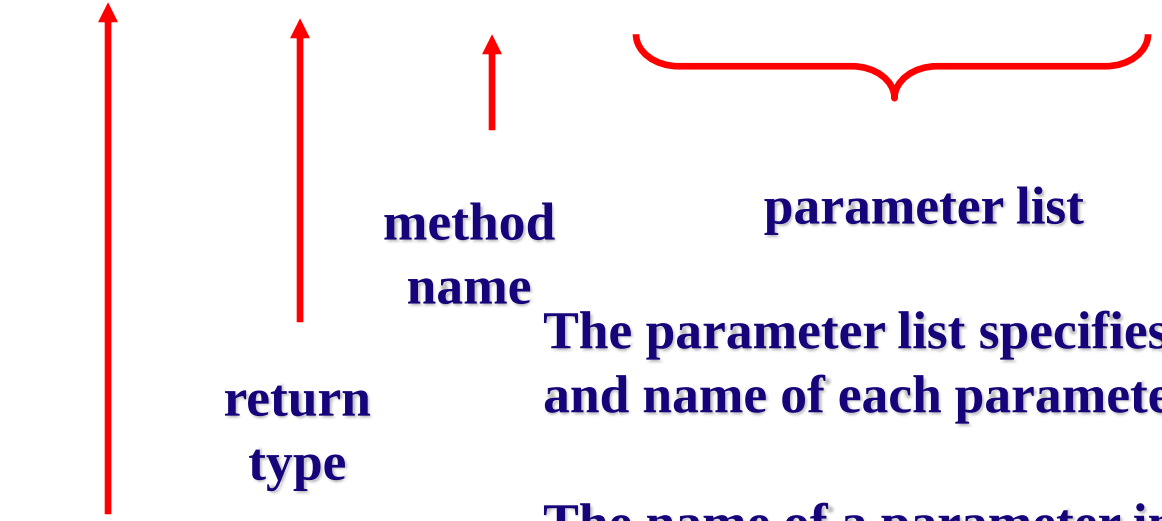
Access_modifiers Return_type method_name (parameter_list)

- **Three parts of a method**
  - **Method declaration**
  - **Method implementation**
  - **Method calling**

METHOD DECLARATION: HEADER

- A method declaration begins with a method header

```
class MyClass
{  …
   static  int    min ( int num1, int num2 )
```



properties

return type

method name

parameter list

The parameter list specifies the type and name of each parameter

The name of a parameter in the method declaration is called a *formal argument*

METHOD DECLARATION: BODY

- The header is followed by the method body:

```
class MyClass
{
    …
    static int min(int num1, int num2)
    {
        int minValue = num1 < num2 ? num1 : num2;
        return minValue;
    }

    …
}
```

THE RETURN STATEMENT

- The return type of a method indicates the type of value that the method sends back to the calling location
  - A method that does not return a value has a void return type
  - The return statement specifies the value that will be returned
  - Its expression must conform to the return type

CALLING A METHOD

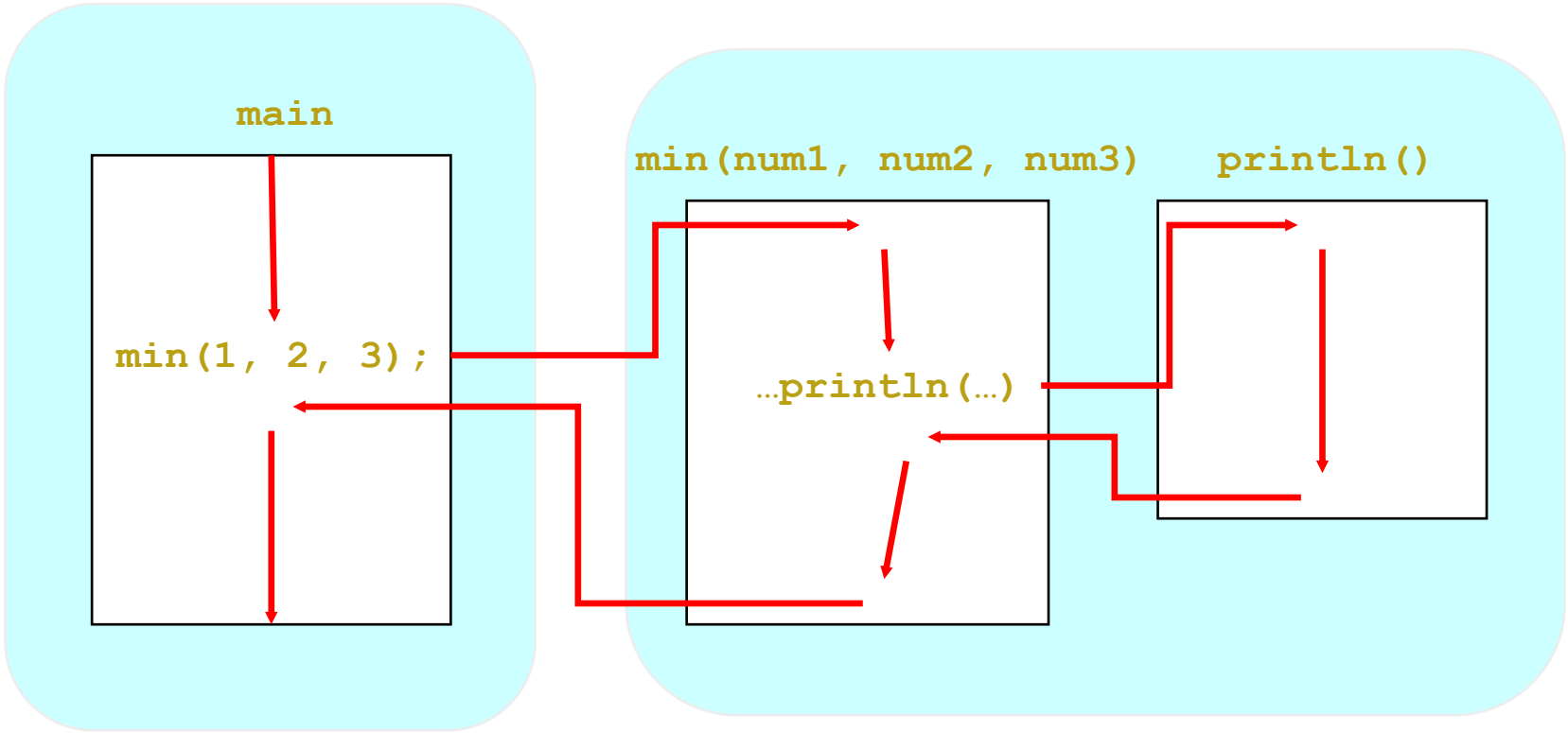
- Each time a method is called, the values of the actual arguments in the invocation are assigned to the formal arguments

```
int  num = min (2, 3);
```

```
static int min (int num1, int num2)
{
   int minValue = (num1 < num2 ? num1 : num2);
   return minValue;
}
```

METHOD CONTROL FLOW

- A method can call another method, who can call another method, …



METHOD :EXAMPLE

Static Method

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

Instance Method

```
public class Main {
     void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

METHOD :EXAMPLE

Static Method

```java
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

```
I just got executed!
```

Instance Method

```java
public class Main {
     void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

```
Main.java:15: error: non-static method myMethod() cannot be referenced from a static context
    myMethod();
    ^
```

METHOD :EXAMPLE

Instance Method

```java
public class Main {
    void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
     Main obj=new Main();
    obj.myMethod();
  }
}
```

```
I just got executed!
```

METHOD OVERLOADING

- A class may define multiple methods with the same name---this is called method overloading usually perform the same task on different data types

- Example: The PrintStream class defines multiple println methods, i.e., println is overloaded:

  println (String s)
  println (int i)
  println (double d)

  . . .

- The following lines use the System.out.print method for different data types:

- System.out.println ("The total is:");
- double total = 0;
- System.out.println (total);

METHOD OVERLOADING: SIGNATURE

- The compiler must be able to determine which version of the method is being invoked
- This is by analyzing the parameters, which form the signature of a method
  - the signature includes the type and order of the parameters
    - if multiple methods match a method call, the compiler picks the best match
    - if none matches exactly but some implicit conversion can be done to match a method, then the method is invoke with implicit conversion.
  - **the return type of the method is not part of the signature**

METHOD OVERLOADING

Version 1

```
double tryMe (int x)
{
    return x + .375;
}
```

Version 2

```
double tryMe (int x, double y)
{
    return x * y;
}
```

Invocation

```
result = tryMe (25, 4.32)
```

MORE EXAMPLES

```
double tryMe ( int x )
{
   return x + 5;
}
```

```
double tryMe ( double x )
{
   return x * .375;
}
```

```
double tryMe (double x, int y)
{
   return x + y;
}
```

Which tryMe will be called?

```
tryMe( 1 );

tryMe( 1.0 );

tryMe( 1.0, 2);

tryMe( 1, 2);

tryMe( 1.0, 2.0);
```

THANK YOU
?