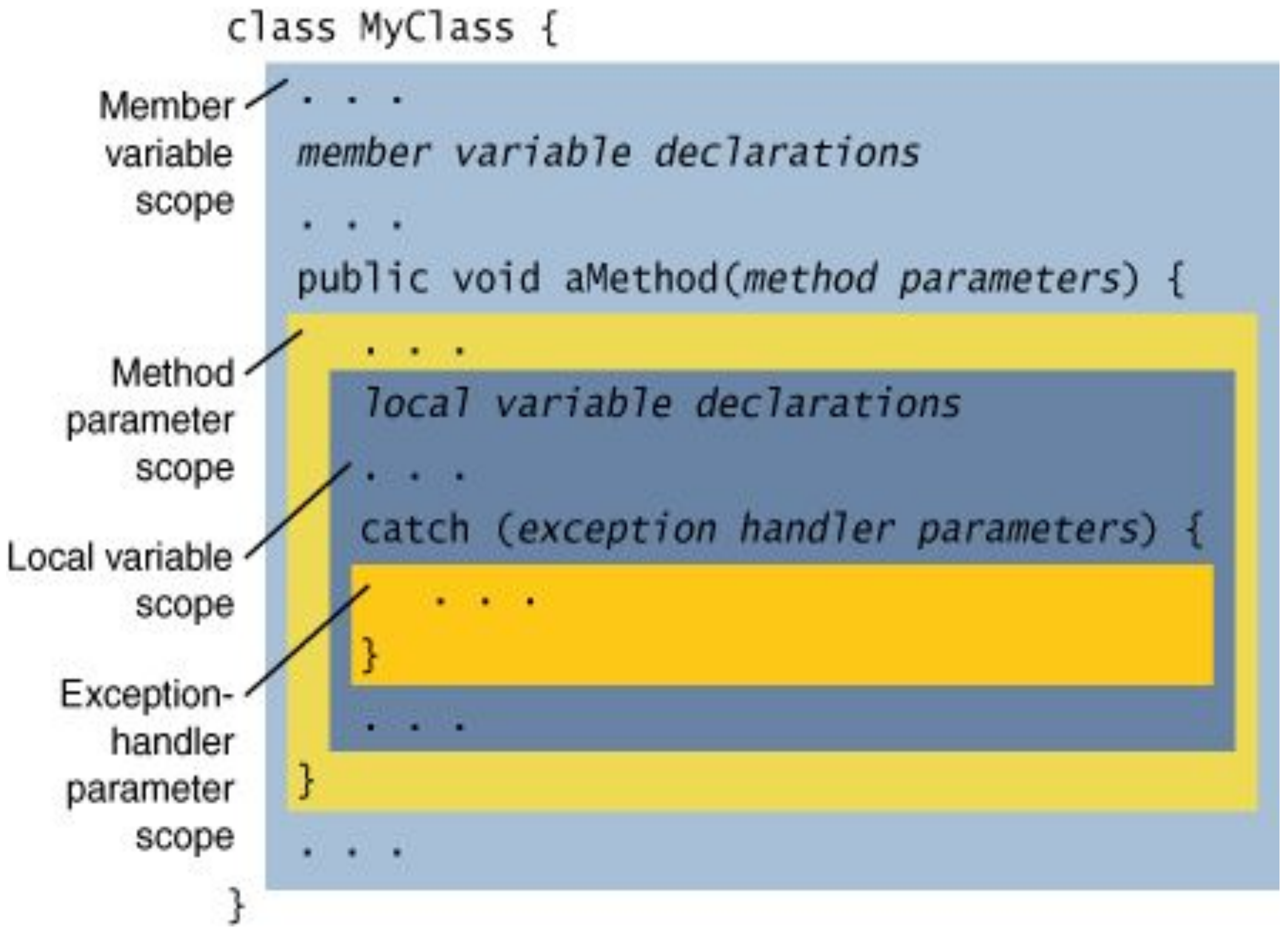
UNIT-II
Building Blocks of Language

Scope of variables

- Java allows variables to be declared within any **block**.
- **A block defines a scope**. Thus, each time you start a new block, you are creating a new scope.
- Scope of a variable determine where the variable is accessible inside or outside the block.
- Java defines two general categories of scopes: **global and local.**
- **Scopes can be nested.** This means that objects declared in the outer scope will be visible to code within the inner scope.

Scope of variables


class MyClass {
Member variable scope
. . .
member variable declarations
. . .
public void aMethod(method parameters) {
Method parameter scope
. . .
local variable declarations
Local variable scope
. . .
catch (exception handler parameters) {
Exception-handler parameter scope
. . .
}
. . .
}
. . .
}

Scope of variables

- **Member variable scope (Class level scope)**
  - These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class.

- **Method parameter scope (Method level scope)**
  - Method parameters accessible only inside that method block. After closing the brackets, new scope is started. So it’s not accessible outside the class.

Scope of variables

- **Local variable scope (Method level scope)**
  - Variables declared inside a method have method level scope and can’t be accessed outside the method.

- **Exception handler parameter scope (Block level scope)**
  - The variables declare in this block, are specifically used for exception handling purpose. And it’s can’t be used after closing exception handling block.

Scope of variables

```
class Scope
{
	public static void main(String args[])
	{
		int x;		// visible to all code within main
		x = 10;

		if(x == 10) // start new scope
		{
			int y = 20;  // Visible only to this block
				// x and y both are visible here.
				System.out.println("x and y: " + x + " " + y);
			x = y * 2;
		}
		y = 100; // Error! y is not visible here (compile-time error)
		// x is still visible here.
		System.out.println("x is " + x);
}
}
```

Default values of variables declared

- If you are not assigning value, then Java runtime assigns default value to variable and when you try to access the variable you get the default value of that variable.

- Following table shows variables types and their default values

Data type	Default value
boolean	FALSE
char	\u0000
int,short,byte / long	0 / 0L
float /double	0.0f / 0.0d
any reference type	null

Type Conversion and Casting

- When you assign value of one data type to another, the two types might not be compatible with each other.
  - If the data types are compatible, then Java will perform the conversion automatically known as **Automatic Type Conversion(Implicit type conversion).**
  - If data types are not compatible, then they need to be casted or converted explicitly, that is known as **Explicit type conversion.**

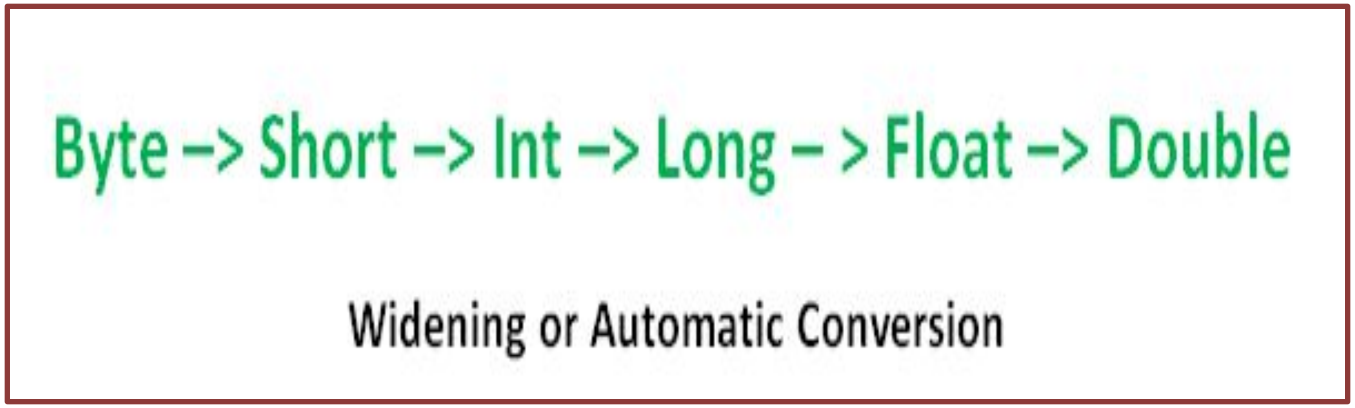
Implicit Type Conversion

- **Widening or Automatic Type Conversion**

- Widening conversion takes place when two data types are automatically converted. This happens when:
  - The two data types are compatible.
  - When we assign value of a smaller data type to a bigger data type.

Implicit Type Conversion

- **Example:** In java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.


Byte –> Short –> Int –> Long – > Float –> Double
Widening or Automatic Conversion

Implicit Type Conversion

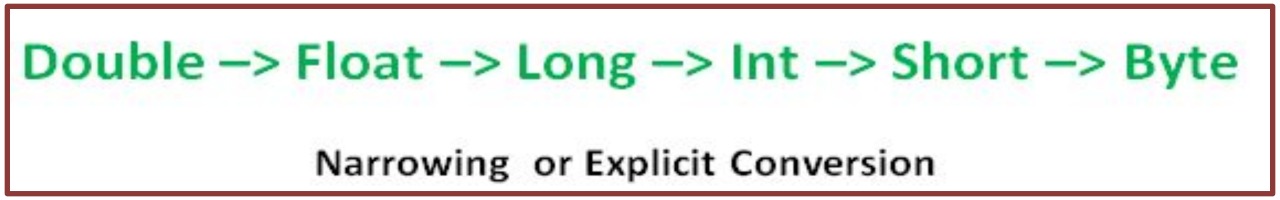
```
class Test
{
    public static void main(String[] args)
    {
        int i = 100;

        //automatic type conversion
        long l = i;

        //automatic type conversion
        float f = l;
        System.out.println("Int value "+i);
        System.out.println("Long value "+l);
        System.out.println("Float value "+f);
    }
}
```

Explicit Type Conversion

- **Narrowing Conversion**
- If we want to assign a value of **larger data type to a smaller data type,** we perform explicit type casting or narrowing.
  - This is useful for incompatible data types where automatic conversion cannot be done.
  - Here, target-type specifies the desired type to convert the specified value.


Double –> Float –> Long –> Int –> Short –> Byte
Narrowing or Explicit Conversion

Explicit Type Conversion

- To create a conversion between **two incompatible** types, you must use a **casting operator**.
- **Syntax:**

  (target-type) value

- Here, **target-type** specifies the desired type to convert the specified value to.

Explicit Type Conversion

- A different type of conversion will occur when a floating-point value is assigned to an integer type: **truncation.**

- **Example:** If the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.

Explicit Type Conversion

```
class Test
{
   public static void main(String[] args)
   {
      double d = 100.04;

      //explicit type casting
      long l = (long)d;
      int i = (int)l;

        System.out.println("Double value "+d);

      //fractional part lost
      System.out.println("Long value "+l);

      //fractional part lost
      System.out.println("Int value "+i);
   }
}
```