Files

Why do we need files

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
- You can easily move your data from one computer to another without any changes.

Type of files

- A file is a stream of bytes
- Two types of files
  - If each byte represents an ascii value, it is a text file
  - Otherwise binary file

Working with files

- C provides functions for all required operations on files
- When working with files, you need to declare a pointer of type file.

```
FILE *fptr;
```

- This declaration is needed for communication between the file and the program.
- File processing operations:
  - Opening/Creating a file
  - Reading the content of the file
  - Writing to a file
  - Moving to specific position within the file
  - Closing the file

File representation

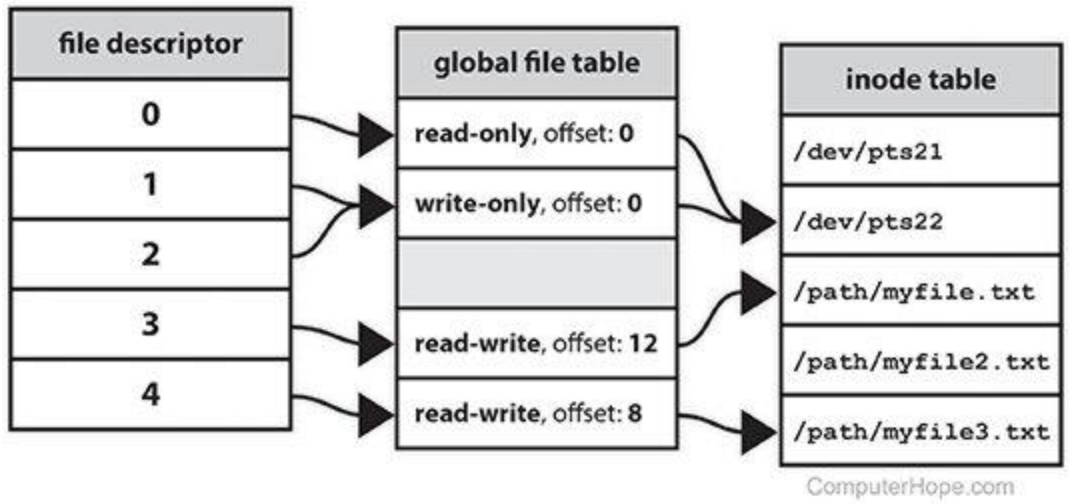
- In C, file is represented as a pointer to structure FILE
  - FILE * pMyFile
- FILE structure includes
  - File descriptor

    A file descriptor is a number that uniquely identifies an open file in a computer's operating system. It describes a data resource, and how that resource may be accessed.
  - File Table

    This table records the mode with which the file (or other resource) has been opened: for reading, writing, and possibly other modes. It also indexes into a third table called the inode table that describes the actual underlying files

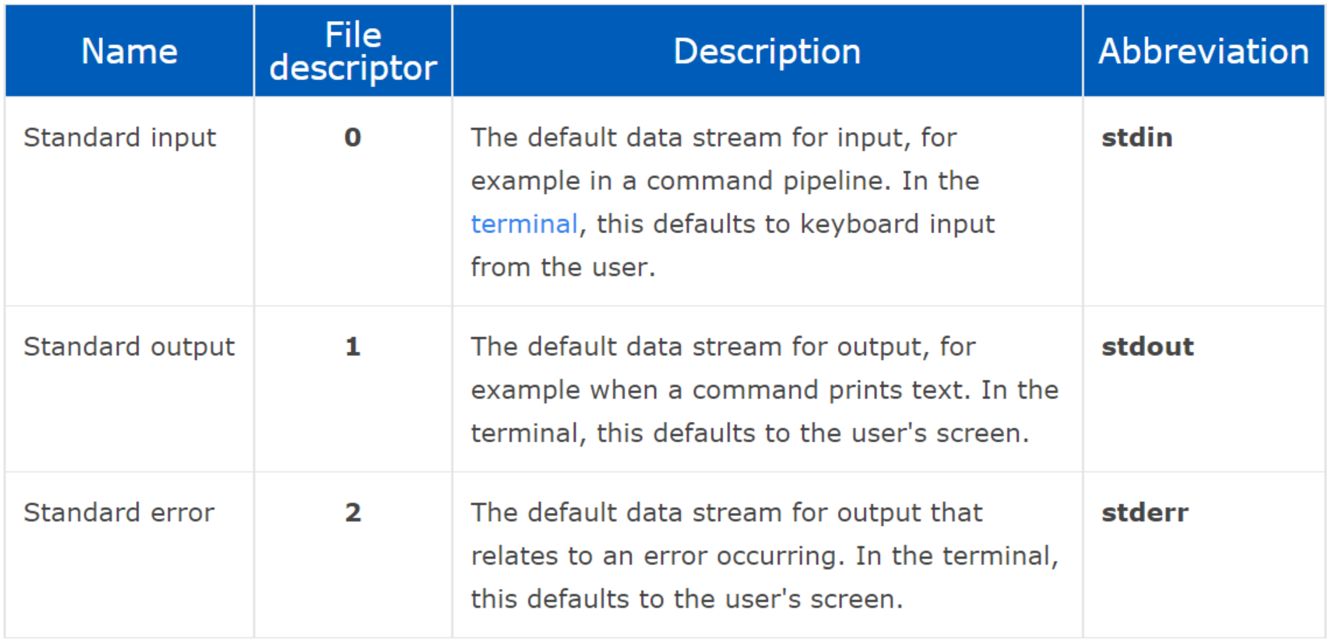
File descriptors/table

- When a process makes a successful request to open a file, the system returns a file descriptor which points to an entry in the global file table. The file table entry contains information such as the inode of the file, byte offset, and the access restrictions for that data stream (read-only, write-only, etc.).



Stdin, Stdout, Stderr

- <stdio.h> contains three standard file pointers that are created (each of type FILE *)
  - stdin - file pointer connected to the keyboard
  - stdout - file pointer connected to the output window/terminal
  - stderr - file pointer connected to the error window (generally the output window)/terminal
- No need to open/close them



Name	File descriptor	Description	Abbreviation
Standard input	**0**	The default data stream for input, for example in a command pipeline. In the terminal, this defaults to keyboard input from the user.	**stdin**
Standard output	**1**	The default data stream for output, for example when a command prints text. In the terminal, this defaults to the user's screen.	**stdout**
Standard error	**2**	The default data stream for output that relates to an error occurring. In the terminal, this defaults to the user's screen.	**stderr**

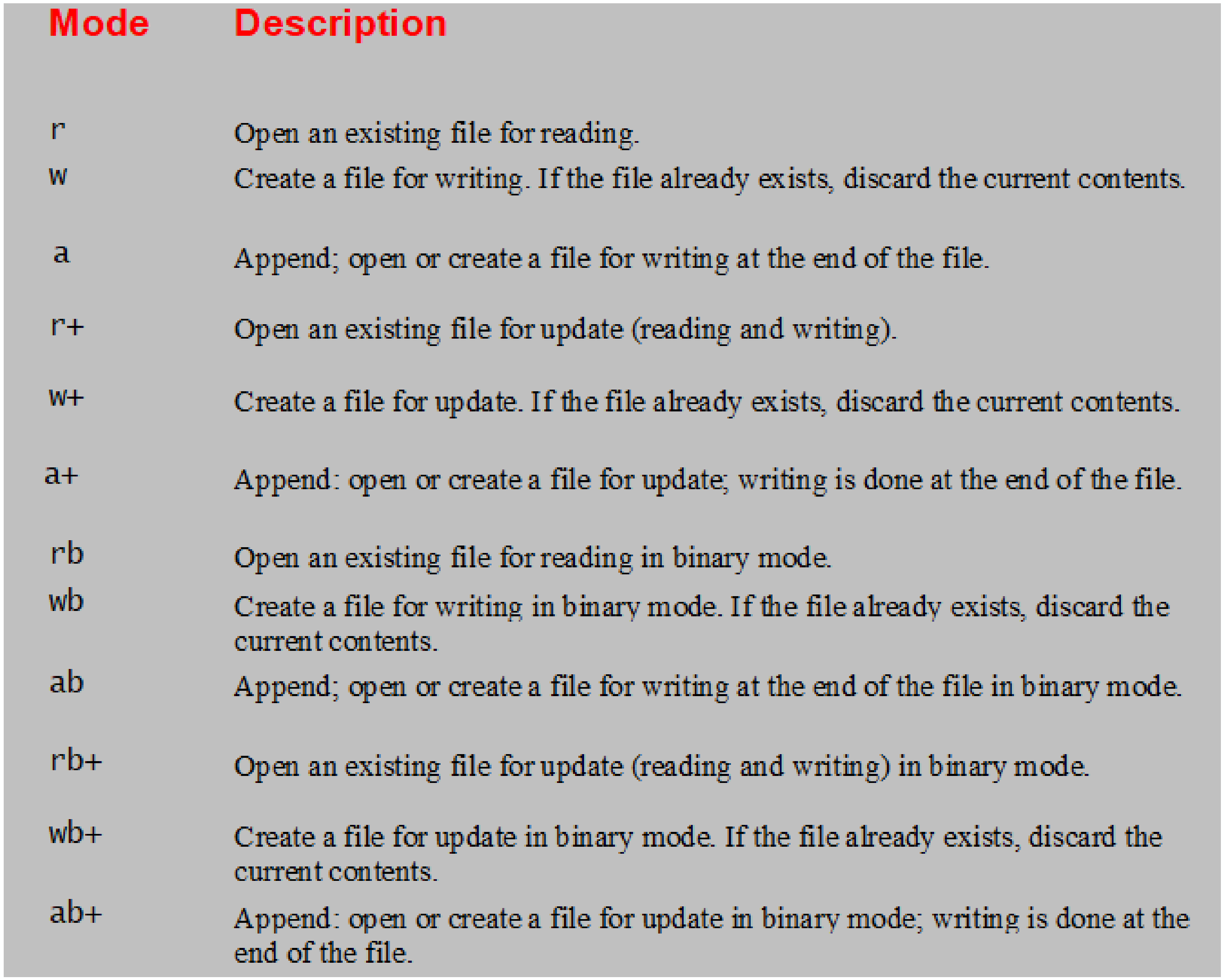
Opening a File

- After creating a file pointer as shown before (FILE *cfPtr)
- Opening a file is performed using the *fopen()* function defined in the stdio.h header file.

```
ptr = fopen("fileopen","mode");
```

- The syntax for opening a file in standard I/O is:
    - cfPtr = fopen(“myfile.dat", “w”);
        - Function fopen returns a file pointer to the specified file
        - Takes two arguments – filename and open mode
        - If open fails, NULL returned
        - Main reasons for failure:
            - File doesn’t exist (read)
            - Can’t create file (append)
- File can be opened in binary mode or text mode
- In text mode newlines and end-of-file are marked by ascii chars. This is later used by functions to read a complete line

Opening a File



Mode	Description
r	Open an existing file for reading.
w	Create a file for writing. If the file already exists, discard the current contents.
a	Append; open or create a file for writing at the end of the file.
r+	Open an existing file for update (reading and writing).
w+	Create a file for update. If the file already exists, discard the current contents.
a+	Append: open or create a file for update; writing is done at the end of the file.
rb	Open an existing file for reading in binary mode.
wb	Create a file for writing in binary mode. If the file already exists, discard the current contents.
ab	Append; open or create a file for writing at the end of the file in binary mode.
rb+	Open an existing file for update (reading and writing) in binary mode.
wb+	Create a file for update in binary mode. If the file already exists, discard the current contents.
ab+	Append: open or create a file for update in binary mode; writing is done at the end of the file.

Closing a File

- fclose(*FilePointer*)

```
fclose(fptr);
```

- The file pointer must be one that was created previously by fopen (and the file remains open)
- fclose returns
  - 0 if the fclose command is successful
  - Special value 'EOF' if the fclose command is unsuccessful

Open/Closing File – Typical code

```
int main() {
  FILE *stream;
  if ((stream = fopen("myfile.dat","r")== NULL) {
    printf("Unable to open file\n");
    return(1);
  }
  /* Read data using FILE */
  if (fclose(stream) == EOF) {
    printf("Error closing file\n");
    return(2);
  }
}
```

Reading and Writing to a file

- Text File -
  - For reading and writing to a text file, we use the functions fprintf() and fscanf().
  - They are just the file versions of printf() and scanf().
  - The only difference is that fprint() and fscanf() expects a pointer to the structure FILE.

- Binary File –
  - Functions fread() and fwrite() are used for reading from and writing to a file respectively.

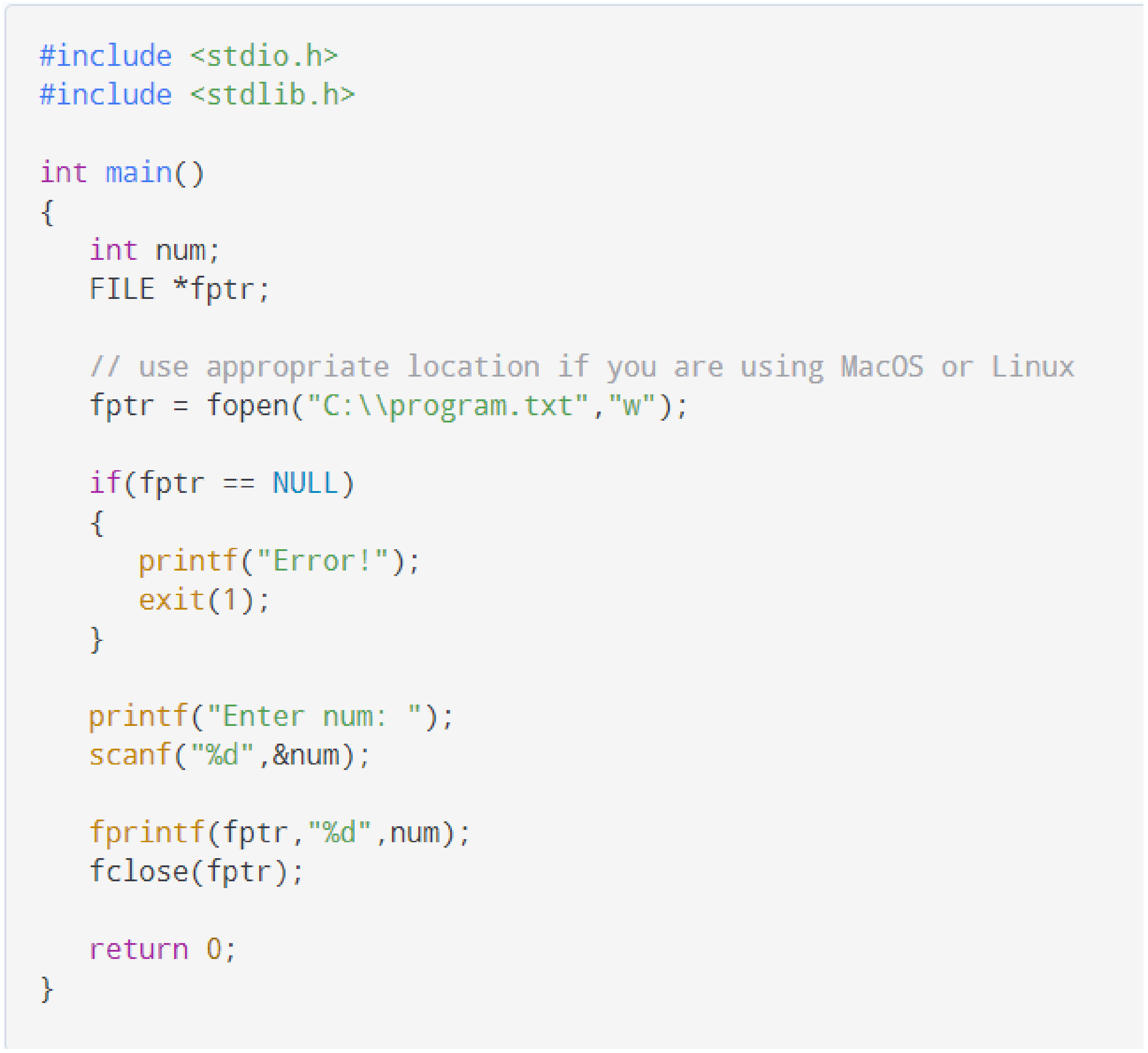
Text Files

- Text files are the normal **.txt** files.
- You can easily create text files using any simple text editors such as Notepad.
- You can easily edit or delete the contents.
- Minimum effort to maintain, easily readable, and provide the least security and takes bigger storage space.
- Often requires larger size than the amount of the data
- Include spaces, new lines descriptors and small number requires several bytes (27 -> 2 chars)

Text file write/read functions

- **fgetc**
  - Reads one character from a file
  - *Syntax- fgetc(FILE *filename)*
  - Equivalent to getchar(stdin), getc(stdin)
- **fputc**
  - Writes one character to a file
  - *Syntax- fputc(int char, FILE *filename)*
  - Equivalent to putchar(), putc(c, stdout)
- **fgets**
  - Reads a line from a file
  - *Syntax- fgets(char *str, int n, FILE *filename). n is the maximal number of chars we want to read*
- **fputs**
  - Writes a string to a file
  - *Syntax- fputs(char *str, FILE *filename)*
- **fscanf / fprintf**
  - File processing equivalents of scanf and printf

Writing to a text file



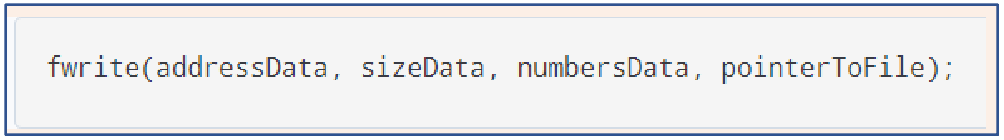
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int num;
   FILE *fptr;

   // use appropriate location if you are using MacOS or Linux
   fptr = fopen("C:\\program.txt","w");

   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }

   printf("Enter num: ");
   scanf("%d",&num);

   fprintf(fptr,"%d",num);
   fclose(fptr);

   return 0;
}
```

Reading from a text file

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.txt","r")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   fscanf(fptr,"%d", &num);

   printf("Value of n=%d", num);
   fclose(fptr);

   return 0;
}
```

Writing to a binary file

- Using *fwrite* function



```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

- *fwrite* syntax
  - Address of data to be written in the disk
  - Size of data to be written in the disk
  - Number of such type of data
  - Pointer to the file where you want to write.

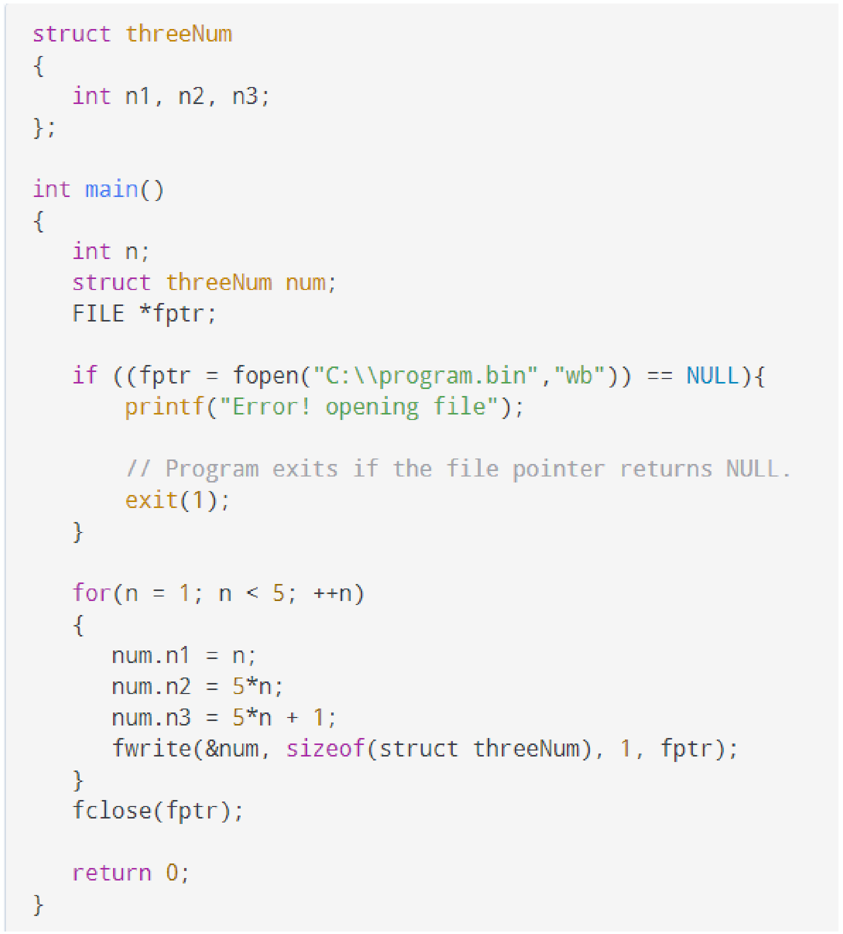
Reading from a binary file

- *fread* function has 4 input arguments, similar to the *fwrite* one -

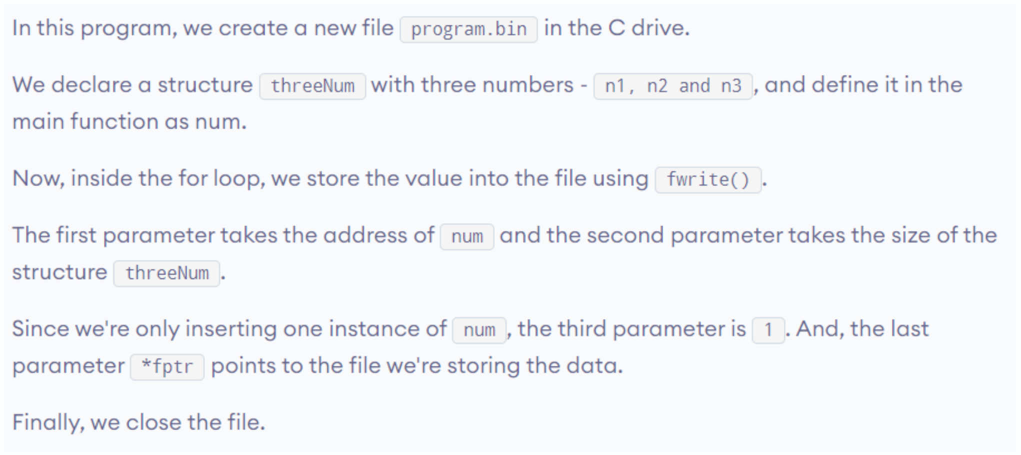
```
fread(addressData, sizeData, numbersData, pointerToFile);
```

- *fread* syntax
  - Address of data to be written in the disk
  - Size of data to be written in the disk
  - Number of such type of data
  - Pointer to the file where you want to write.

Writing to a binary file



```c
struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   for(n = 1; n < 5; ++n)
   {
      num.n1 = n;
      num.n2 = 5*n;
      num.n3 = 5*n + 1;
      fwrite(&num, sizeof(struct threeNum), 1, fptr);
   }
   fclose(fptr);

   return 0;
}
```



In this program, we create a new file `program.bin` in the C drive.

We declare a structure `threeNum` with three numbers - `n1, n2 and n3`, and define it in the main function as num.

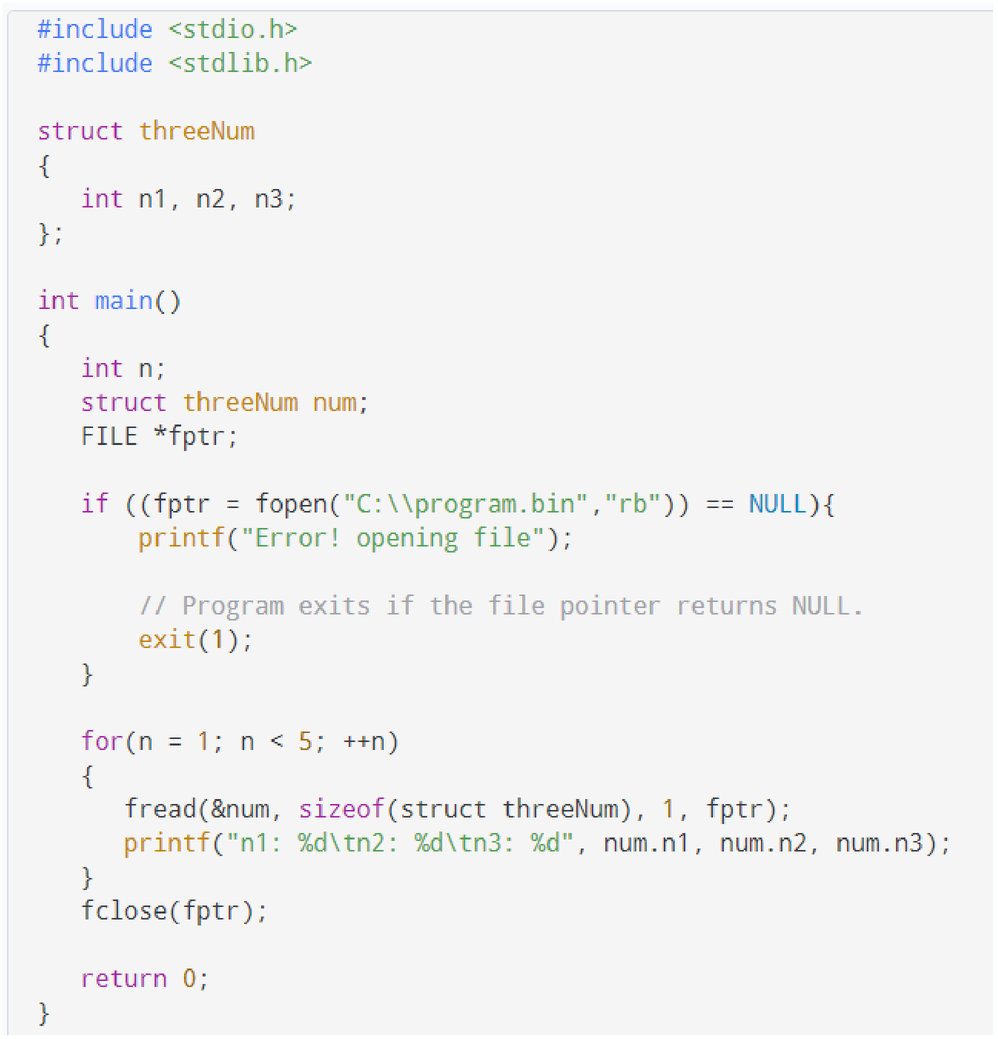
Now, inside the for loop, we store the value into the file using `fwrite()`.

The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.

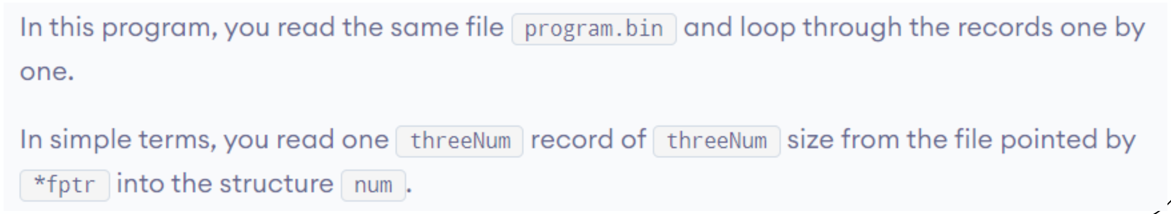
Since we're only inserting one instance of `num`, the third parameter is `1`. And, the last parameter `*fptr` points to the file we're storing the data.

Finally, we close the file.

Reading from a binary file



```c
#include <stdio.h>
#include <stdlib.h>

struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
   }
   fclose(fptr);

   return 0;
}
```



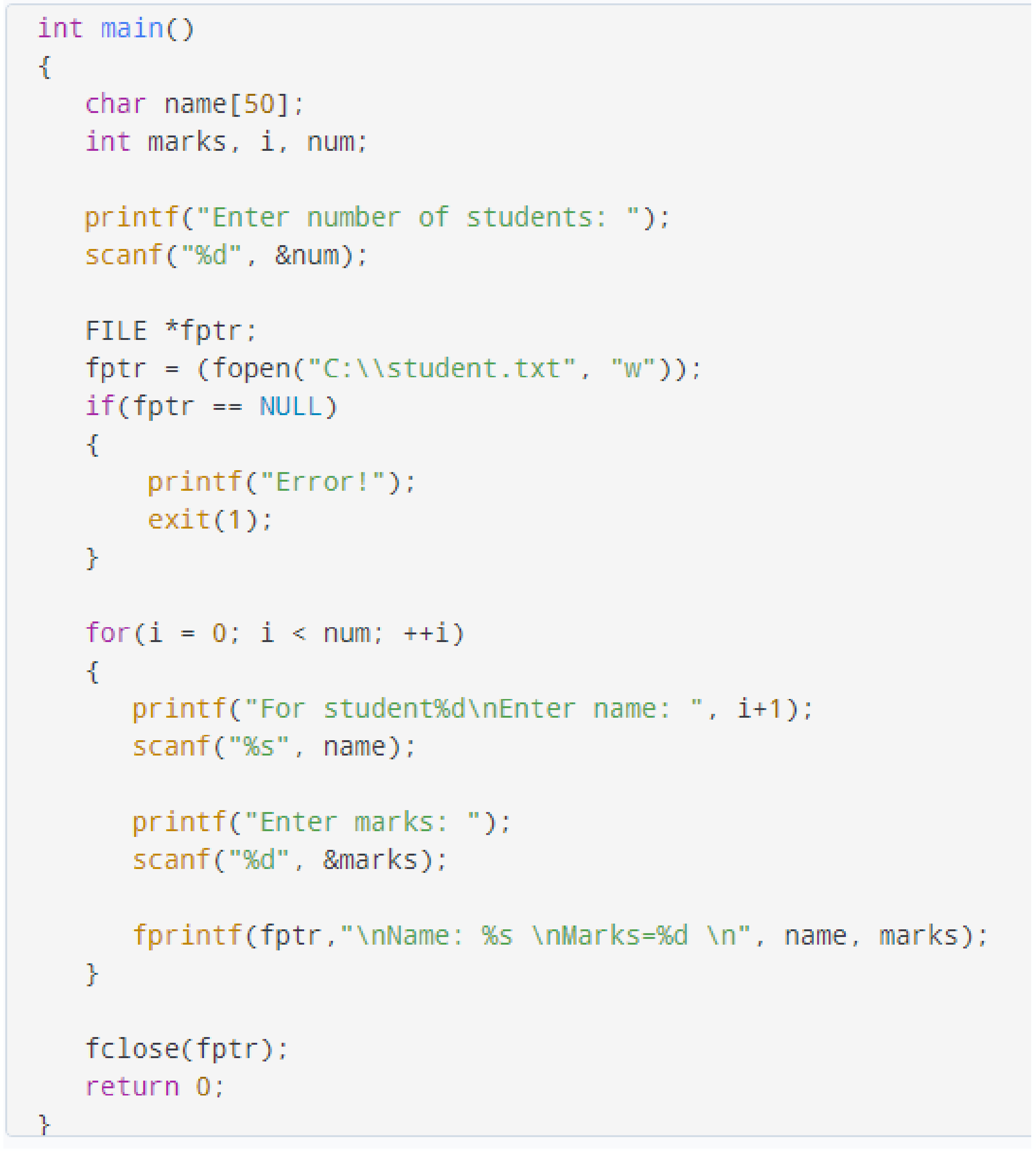
In this program, you read the same file `program.bin` and loop through the records one by one.

In simple terms, you read one `threeNum` record of `threeNum` size from the file pointed by `*fptr` into the structure `num`.

Example 1

Write C program to read names and marks of n number of students and store them in a file

Example 1



```c
int main()
{
   char name[50];
   int marks, i, num;

   printf("Enter number of students: ");
   scanf("%d", &num);

   FILE *fptr;
   fptr = (fopen("C:\\student.txt", "w"));
   if(fptr == NULL)
   {
       printf("Error!");
       exit(1);
   }

   for(i = 0; i < num; ++i)
   {
      printf("For student%d\nEnter name: ", i+1);
      scanf("%s", name);

      printf("Enter marks: ");
      scanf("%d", &marks);

      fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
   }

   fclose(fptr);
   return 0;
}
```

Example 2

Write a C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.

Example 2

```c
#include <stdio.h>
struct student
{
   char name[50];
   int height;
};
int main(){
    struct student stud1[5], stud2[5];
    FILE *fptr;
    int i;

    fptr = fopen("file.txt","wb");
    for(i = 0; i < 5; ++i)
    {
        fflush(stdin);
        printf("Enter name: ");
        gets(stud1[i].name);

        printf("Enter height: ");
        scanf("%d", &stud1[i].height);
    }

    fwrite(stud1, sizeof(stud1), 1, fptr);
    fclose(fptr);

    fptr = fopen("file.txt", "rb");
    fread(stud2, sizeof(stud2), 1, fptr);
    for(i = 0; i < 5; ++i)
    {
        printf("Name: %s\nHeight: %d", stud2[i].name, stud2[i].height);
    }
    fclose(fptr);
```

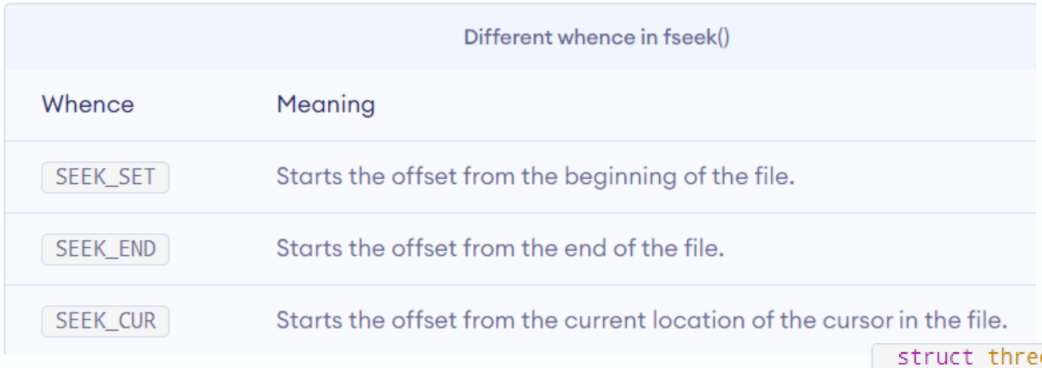
Move "cursor" inside a file

- If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the desired one. This will waste a lot of memory and operation time.
- An easier way to get to the required data can be achieved using *fseek()*
- As the name suggests, *fseek()* seeks the cursor to the given record in the file
- The syntax –

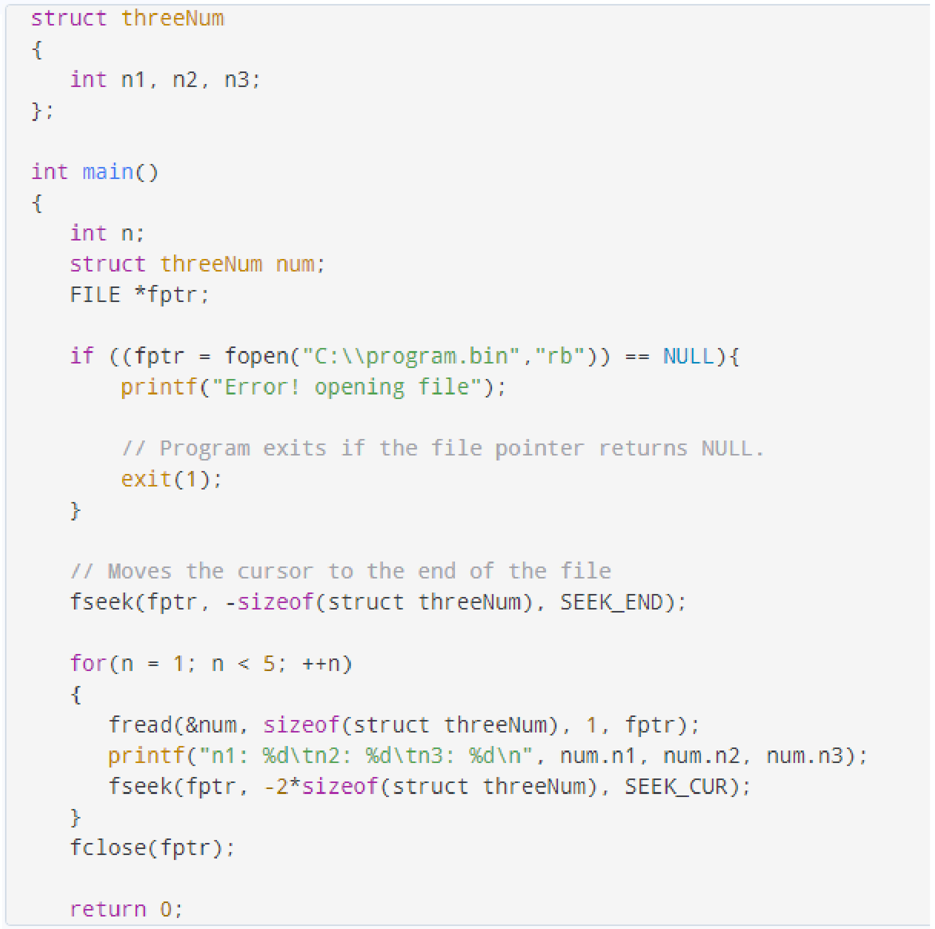
```
fseek(FILE * stream, long int offset, int whence);
```

- The first parameter stream is the pointer to the file. The second parameter is the number of bytes to offset from whence, and the third parameter is the position from where offset is added. It is specified by one of the following constants -

fseak a binary file



Different whence in fseek()	
Whence	**Meaning**
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.



```c
struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   // Moves the cursor to the end of the file
   fseek(fptr, -sizeof(struct threeNum), SEEK_END);

   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
      fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
   }
   fclose(fptr);

   return 0;
```

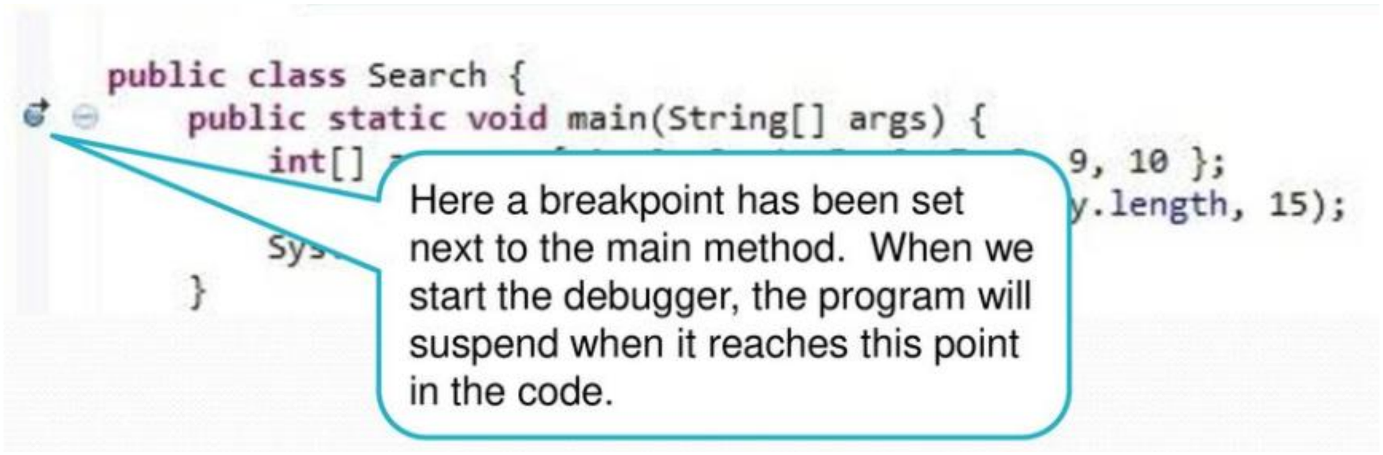
IDE Debugging

Eclipse IDE Debugger

- Eclipse contains a powerful debugger that enables to easily find and fix problems
- It lets us suspend a program during execution, in order to allow the user to step through the code line by line
- We will demonstrate the basic functionality of the debugger first and then debug a buggy program

Breakpoints

- Before we can begin debugging, we need to set a breakpoint
- Whenever the debugger encounters a breakpoint, the program will be suspended to allow the user to start debugging
- To set a breakpoint, double click on the left margin in the editor window.
- To remove a breakpoint, double click on it again
- If no breakpoint is set, the debugger will automatically suspend only if it encounters an exception during runtime

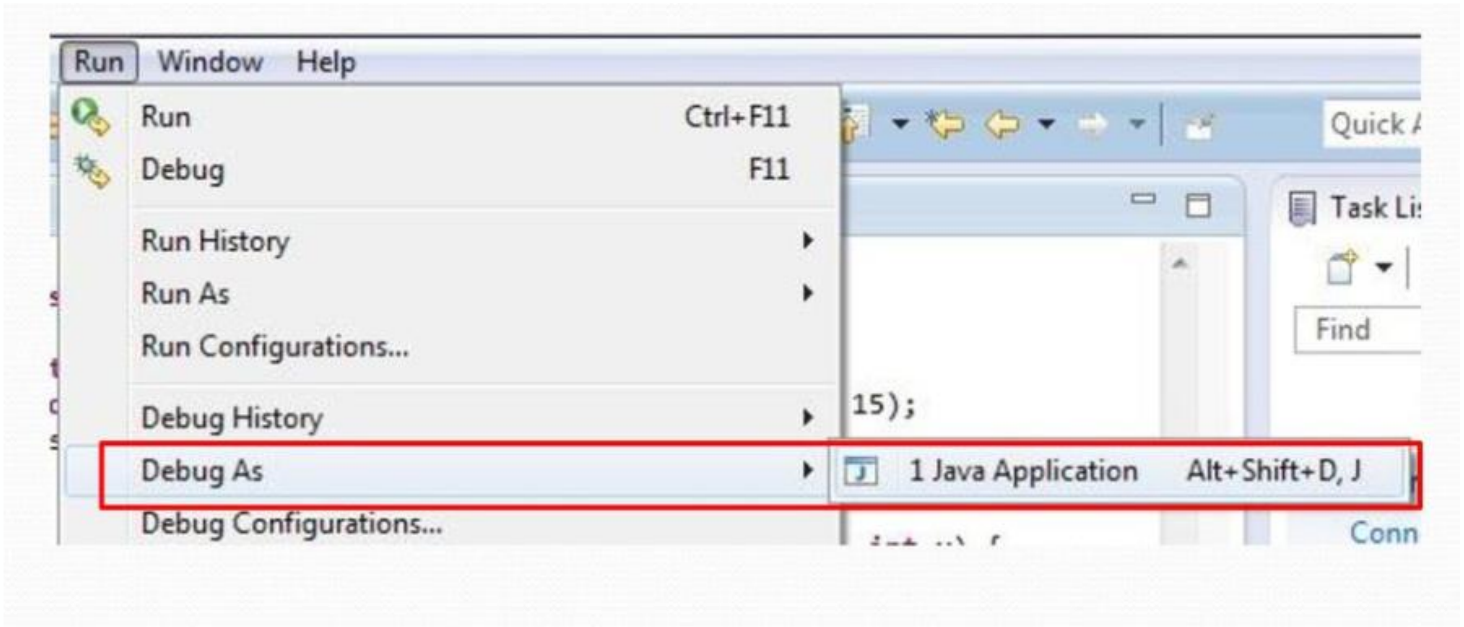


```
public class Search {
    public static void main(String[] args) {
        int[] ...                                9, 10 };
                                                y.length, 15);
        Sys...
    }
```

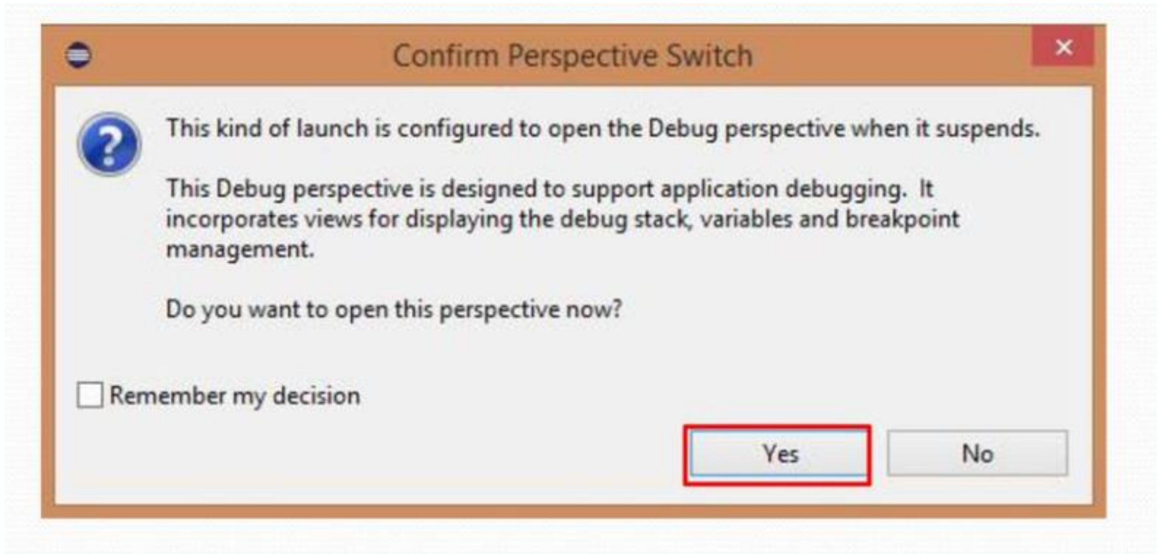
Here a breakpoint has been set next to the main method. When we start the debugger, the program will suspend when it reaches this point in the code.

Running the Debugger

- To run the debugger, select – Run → Debug As → C application
- This brings Eclipse to the debug perspective



- If the ‘Confirm Perspective Switch’ window appears, click ‘Yes’



This kind of launch is configured to open the Debug perspective when it suspends.

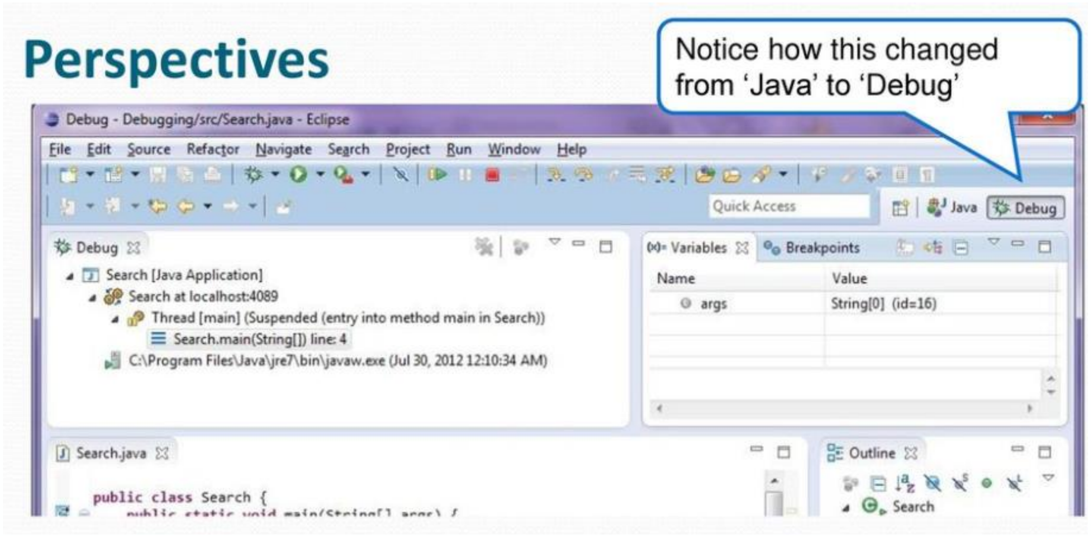
This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

☐ Remember my decision

Perspective

- Now the IDE is in the debug 'perspective'
- A perspective is a set of views and windows. It provides the user with the necessary views and windows for a certain task
- The default perspective that we have seen so far is the 'C' perspective, and is used for coding.
- When Eclipse debugger is run, the workbench automatically switches to the 'Debug' perspective


Perspectives
Notice how this changed from 'Java' to 'Debug'
Debug - Debugging/src/Search.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Debug
Variables
Breakpoints
Name
Value
Search.java
Outline
Search

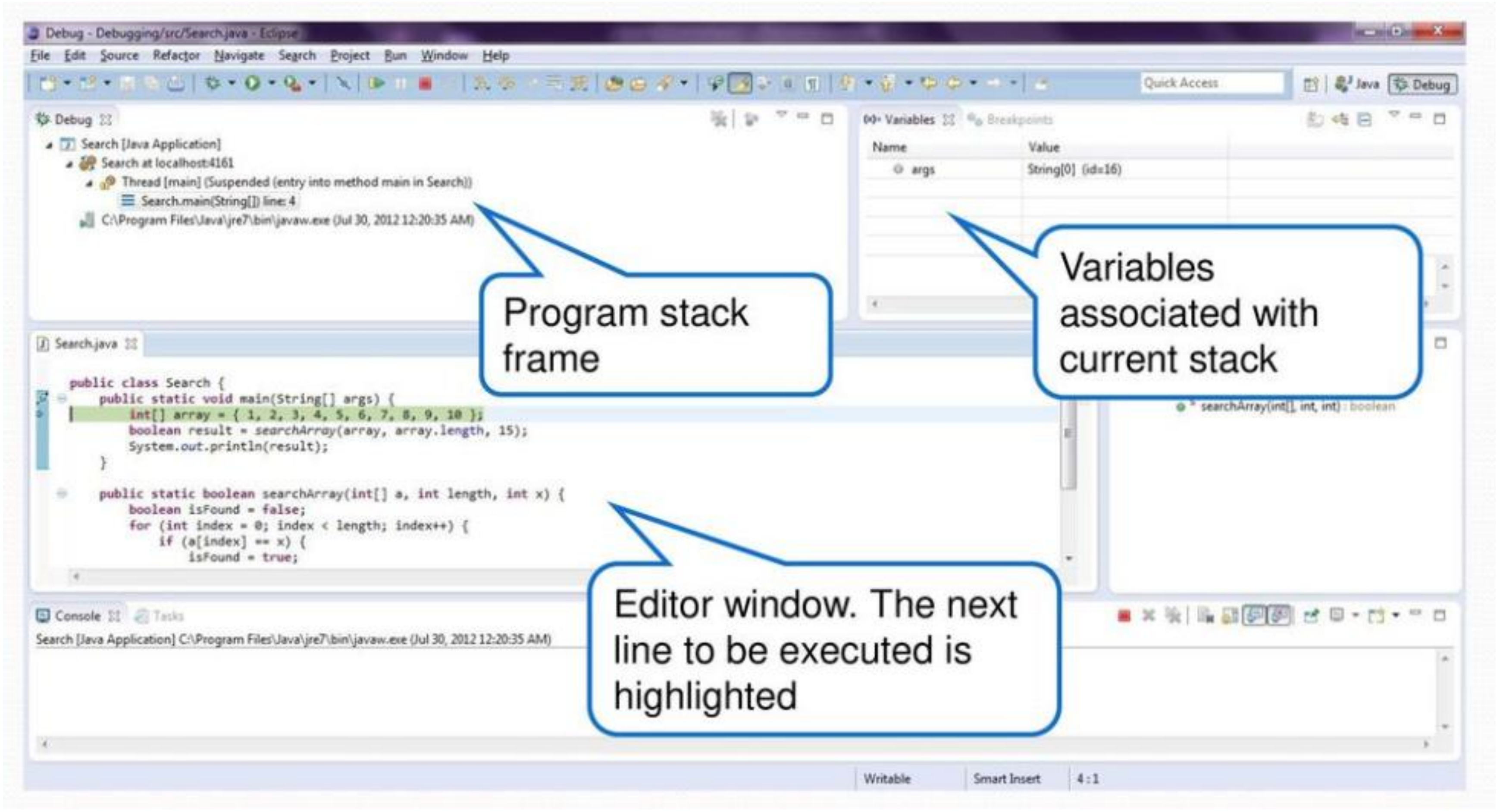
```
Search [Java Application]
  Search at localhost:4089
    Thread [main] (Suspended (entry into method main in Search))
      Search.main(String[]) line: 4
  C:\Program Files\Java\jre7\bin\javaw.exe (Jul 30, 2012 12:10:34 AM)
```

Name	Value
args	String[0] (id=16)

```
public class Search {
```

Debug Perspective

- These are the main components of the debug perspective



Debug - Debugging/src/Search.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Debug
Program stack frame
Variables associated with current stack
Editor window. The next line to be executed is highlighted
Console
Tasks
Writable
Smart Insert
4 : 1

Debug

```
Search [Java Application]
  Search at localhost:4161
    Thread [main] (Suspended (entry into method main in Search))
      Search.main(String[]) line: 4
  C:\Program Files\Java\jre7\bin\javaw.exe (Jul 30, 2012 12:20:35 AM)
```

Variables / Breakpoints

Name	Value
args	String[0] (id=16)

Search.java

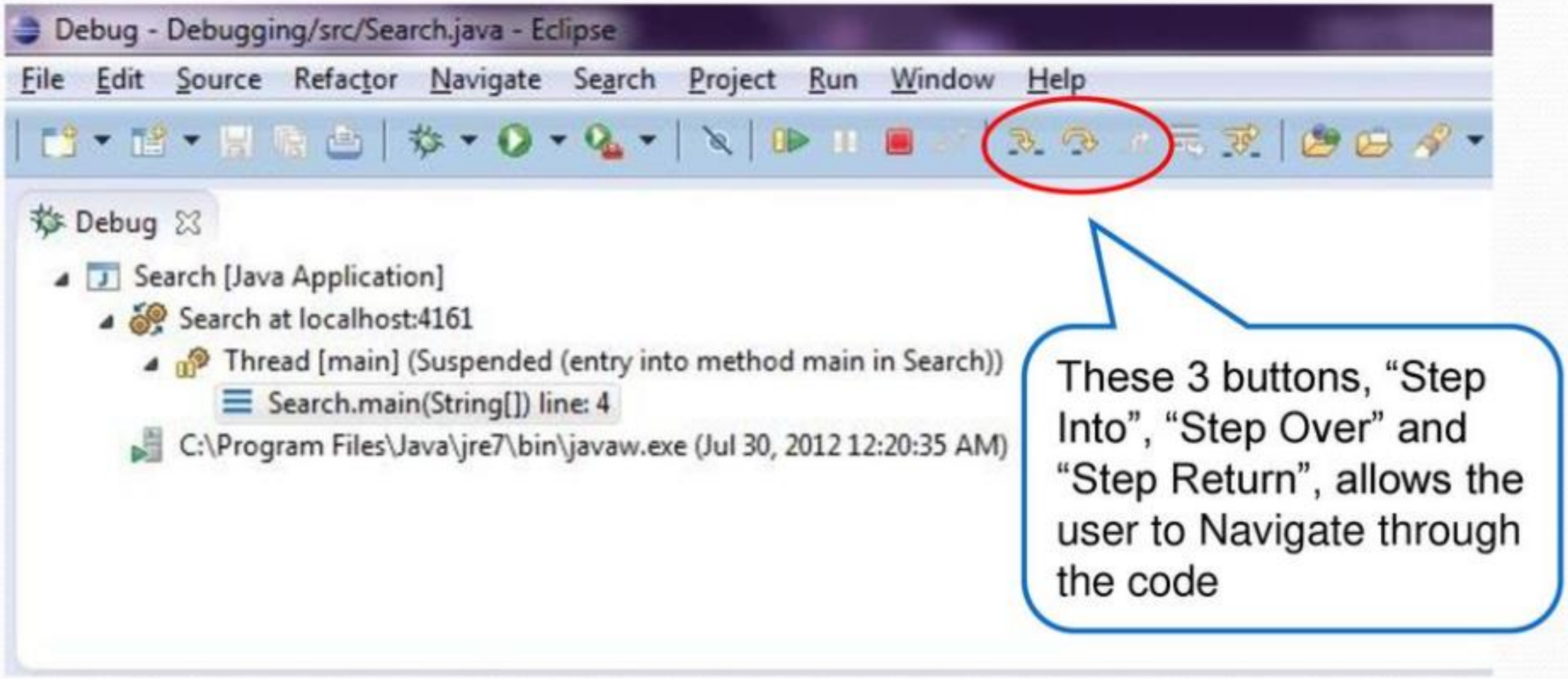
```
public class Search {
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
```

searchArray(int[], int, int) : boolean

Console

```
Search [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jul 30, 2012 12:20:35 AM)
```

Stepping through the code


Debug - Debugging/src/Search.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug

Search [Java Application]
Search at localhost:4161
Thread [main] (Suspended (entry into method main in Search))
Search.main(String[]) line: 4
C:\Program Files\Java\jre7\bin\javaw.exe (Jul 30, 2012 12:20:35 AM)

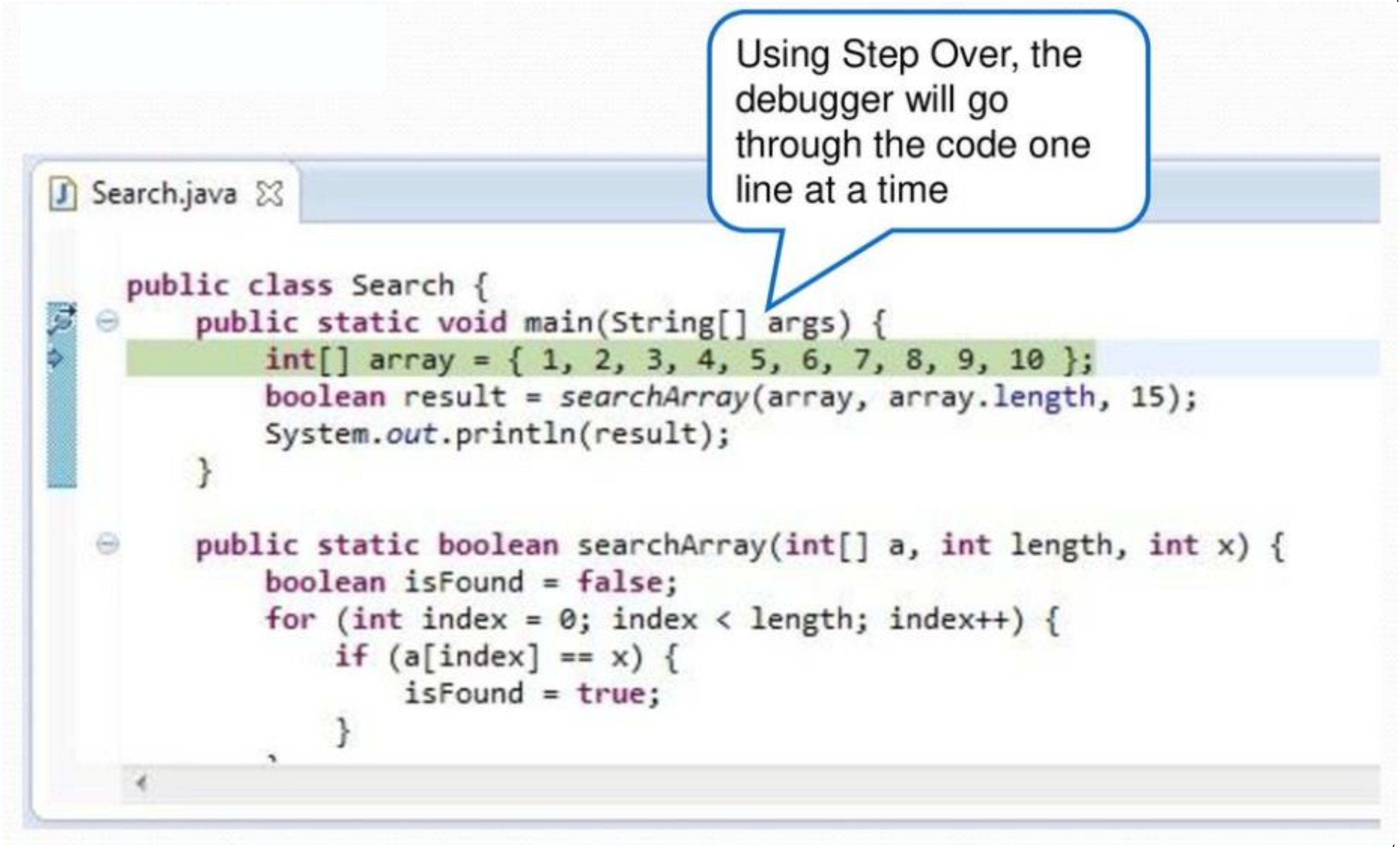
These 3 buttons, "Step Into", "Step Over" and "Step Return", allows the user to Navigate through the code

Step Into (F5) : Moves into the method or constructor at the current line

Step Over (F6): Executes current line and suspend at the next line

Step Return (F7): Executes until the current function returns and pauses immediately

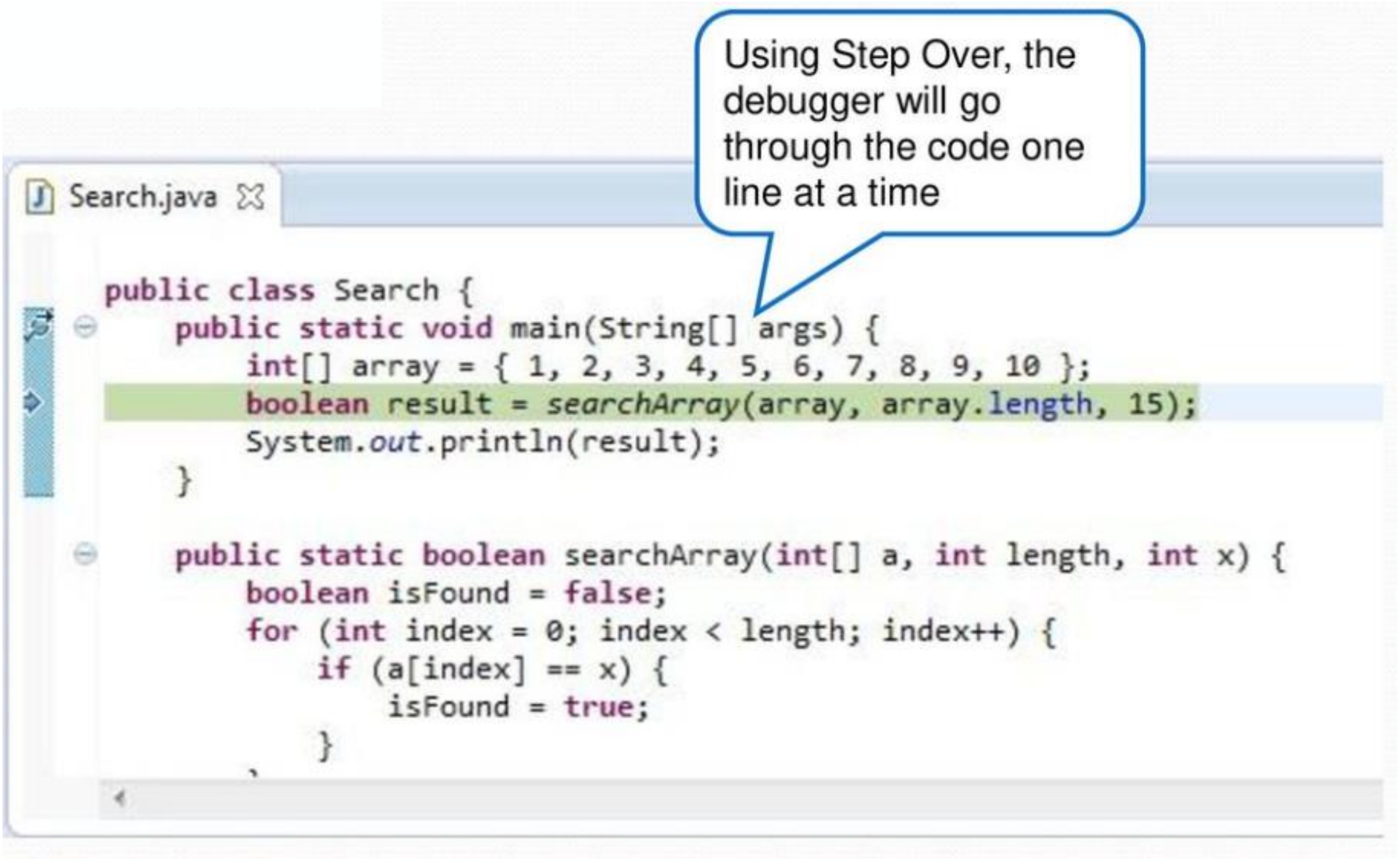
Step Over



Using Step Over, the debugger will go through the code one line at a time

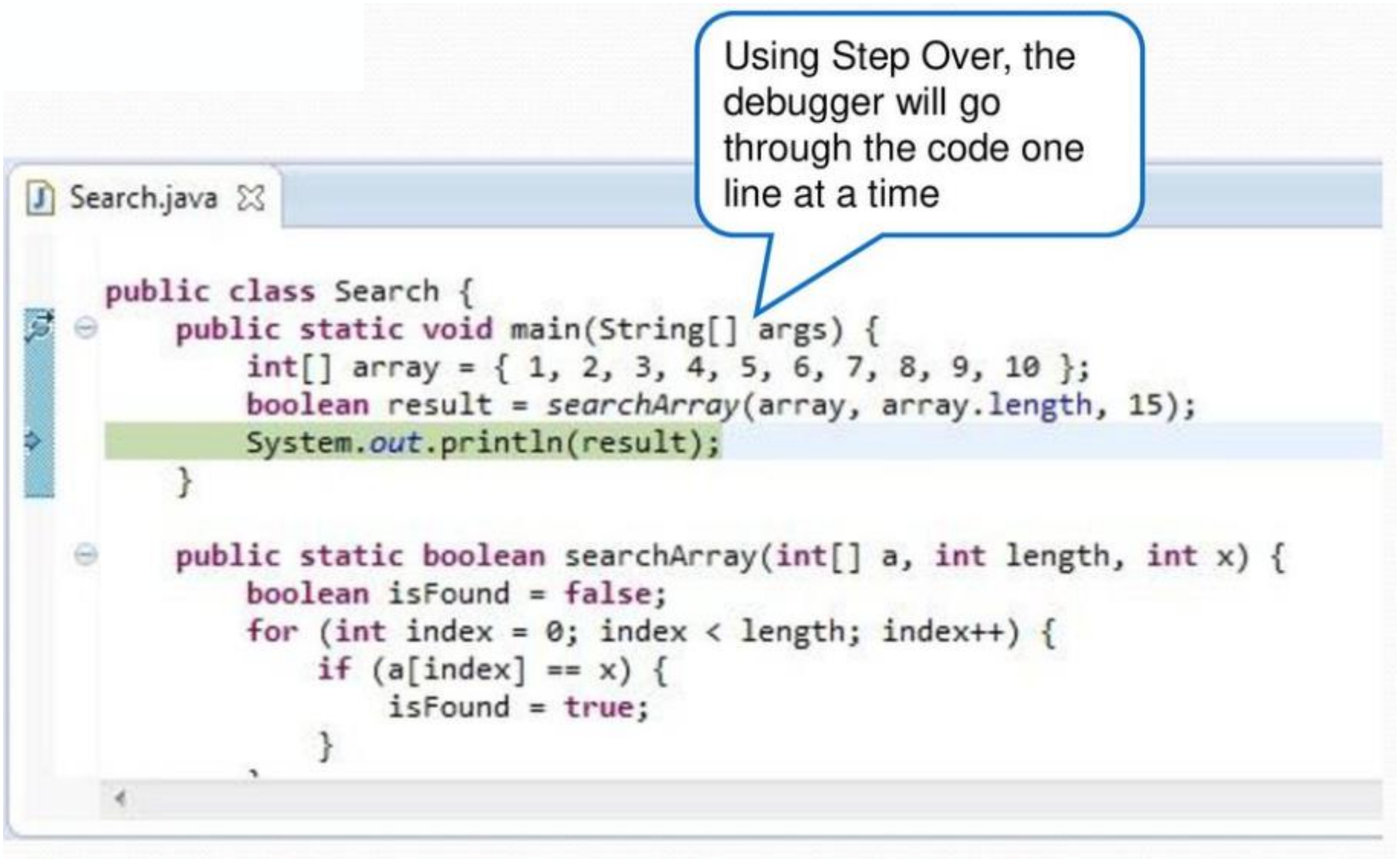
Search.java

```
public class Search {
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```



Using Step Over, the debugger will go through the code one line at a time

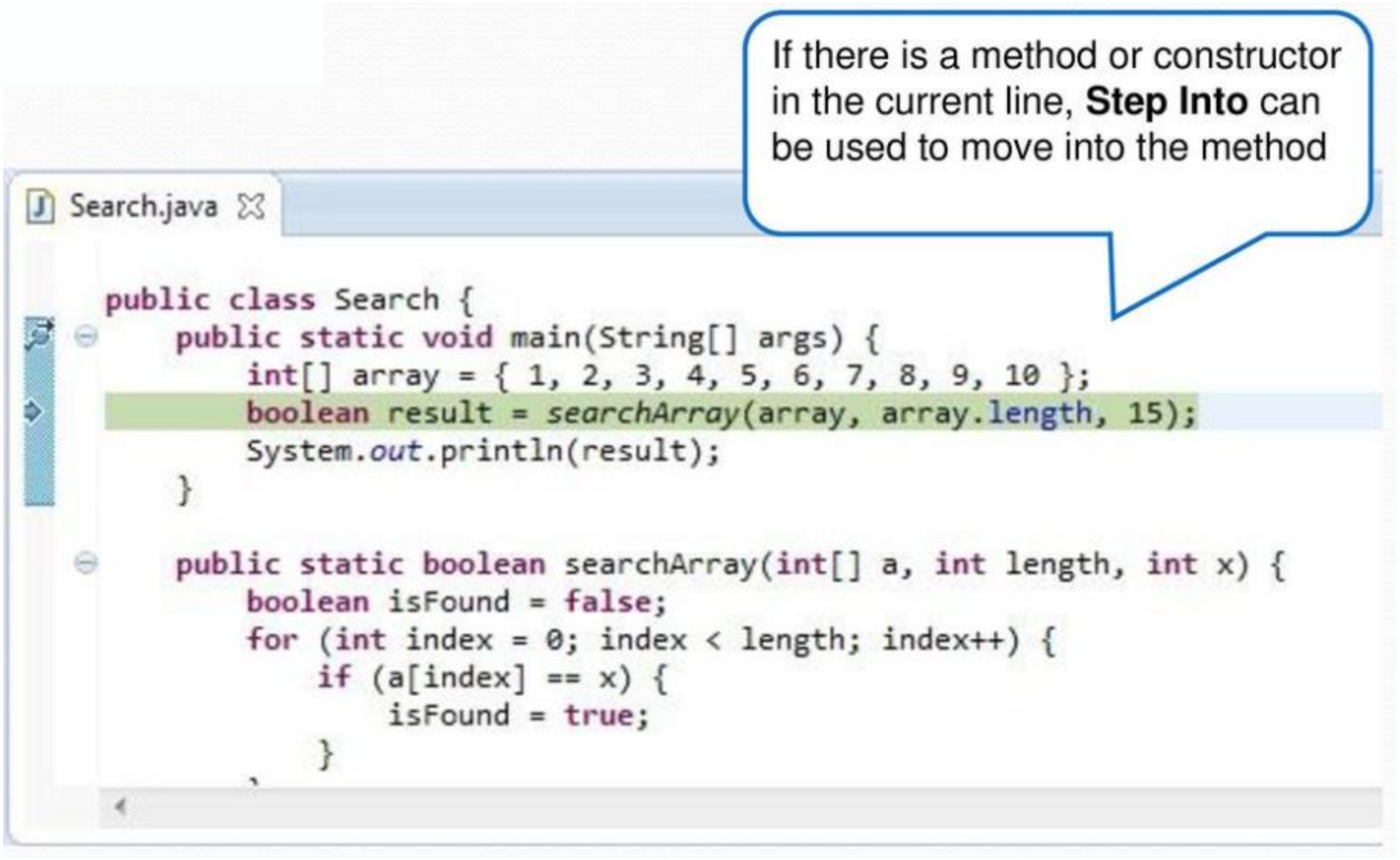
```
public class Search {
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```



```
public class Search {
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```

Using Step Over, the debugger will go through the code one line at a time

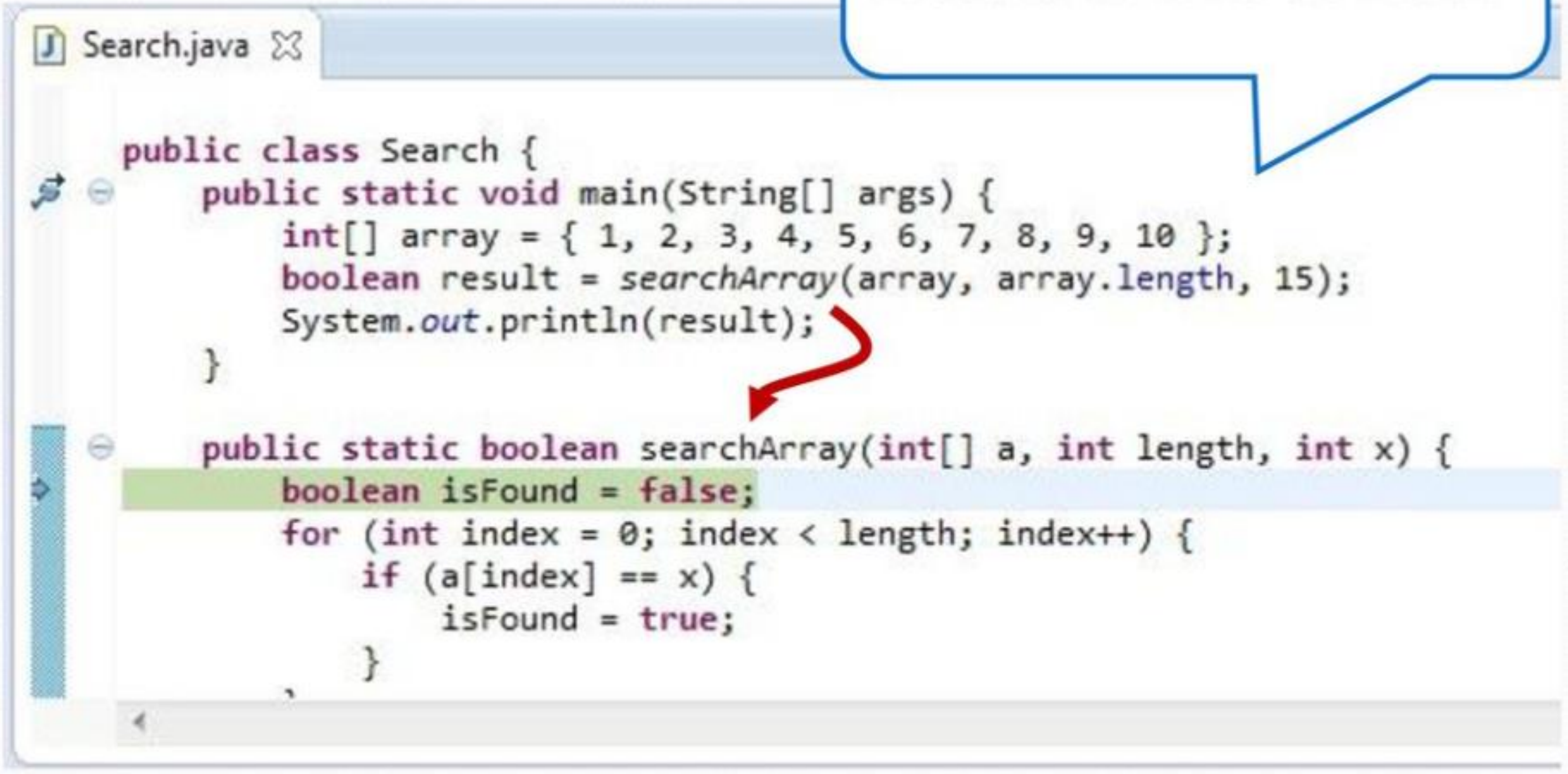
Step Into



If there is a method or constructor in the current line, **Step Into** can be used to move into the method

```
public class Search {
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```

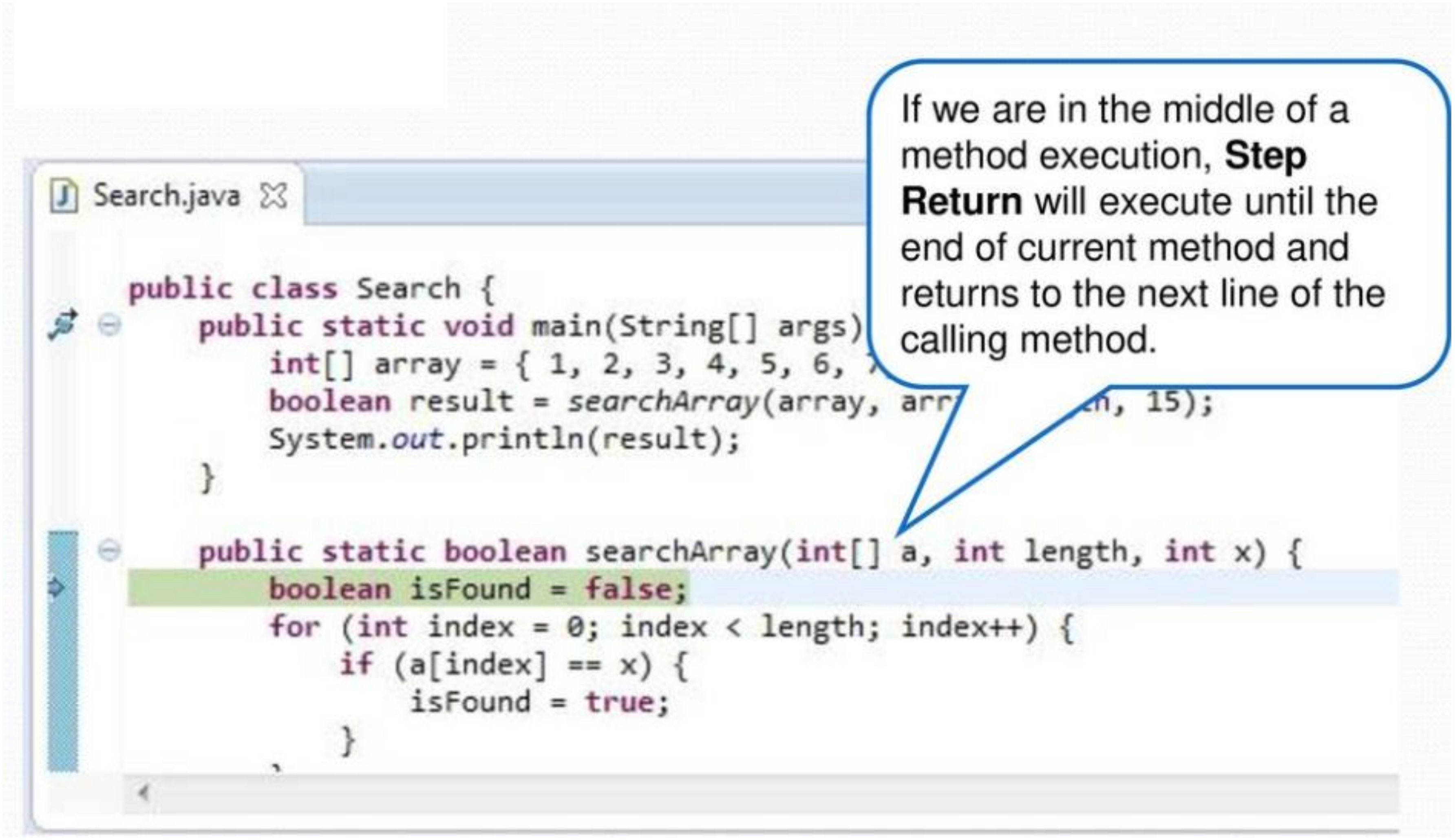
If there is a method or constructor in the current line, **Step Into** can be used to move into the method



Search.java

```
public class Search {
    public static void main(String[] args) {
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```

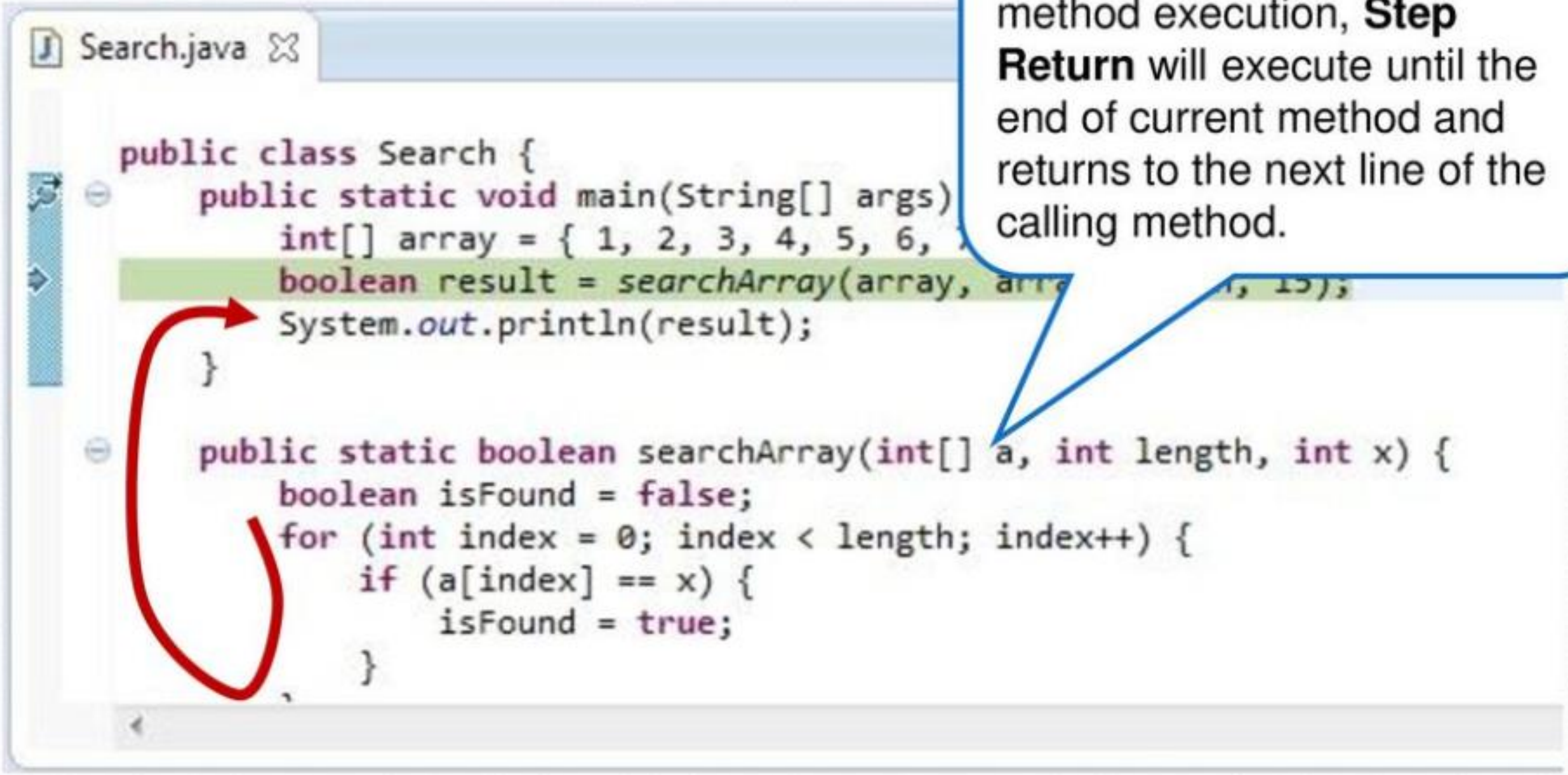
Step return



Search.java

```
public class Search {
    public static void main(String[] args)
        int[] array = { 1, 2, 3, 4, 5, 6, 7
        boolean result = searchArray(array, arr      h, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```

If we are in the middle of a method execution, **Step Return** will execute until the end of current method and returns to the next line of the calling method.

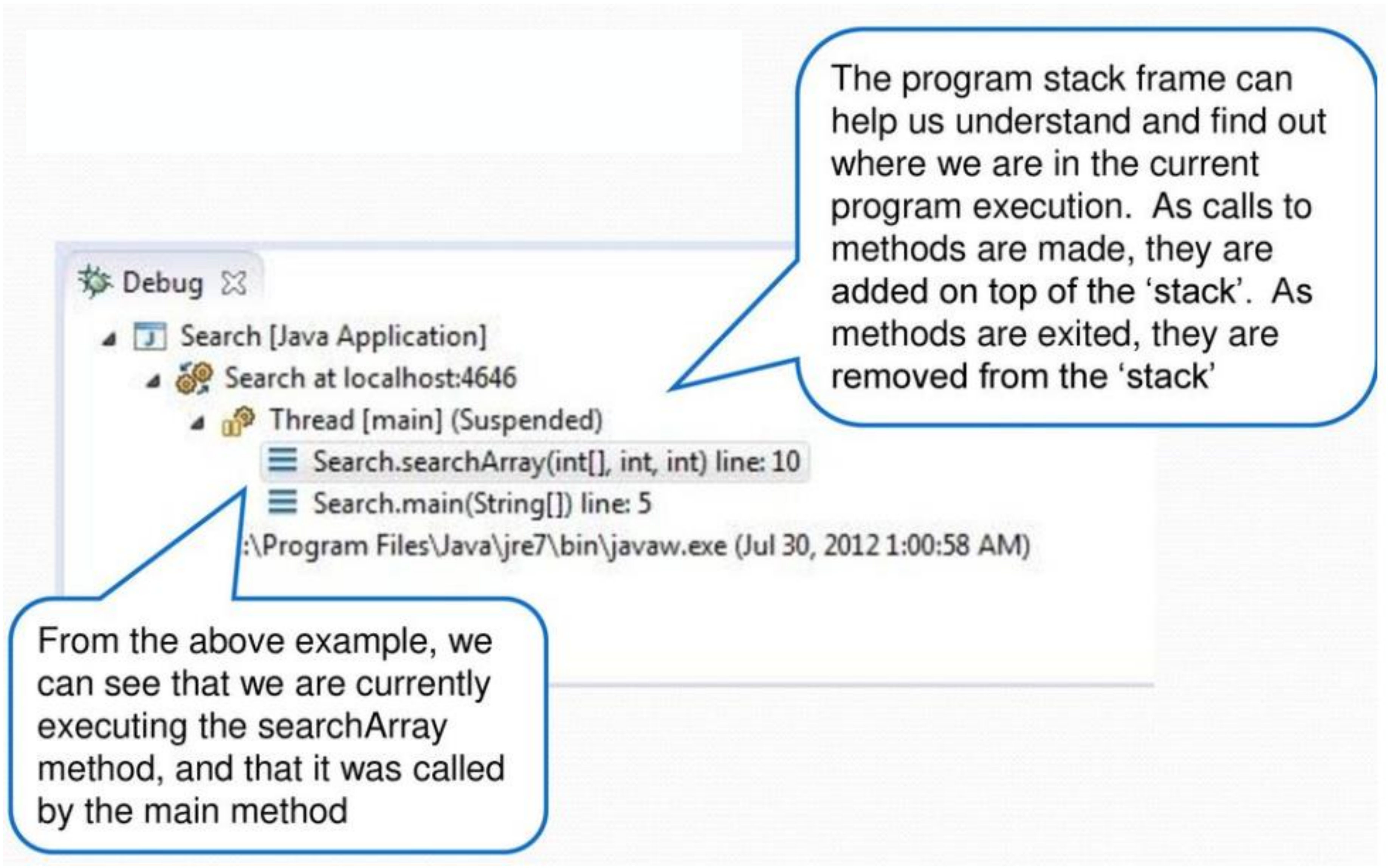


Search.java

```
public class Search {
    public static void main(String[] args)
        int[] array = { 1, 2, 3, 4, 5, 6,
        boolean result = searchArray(array, arr
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
```

If we are in the middle of a method execution, **Step Return** will execute until the end of current method and returns to the next line of the calling method.

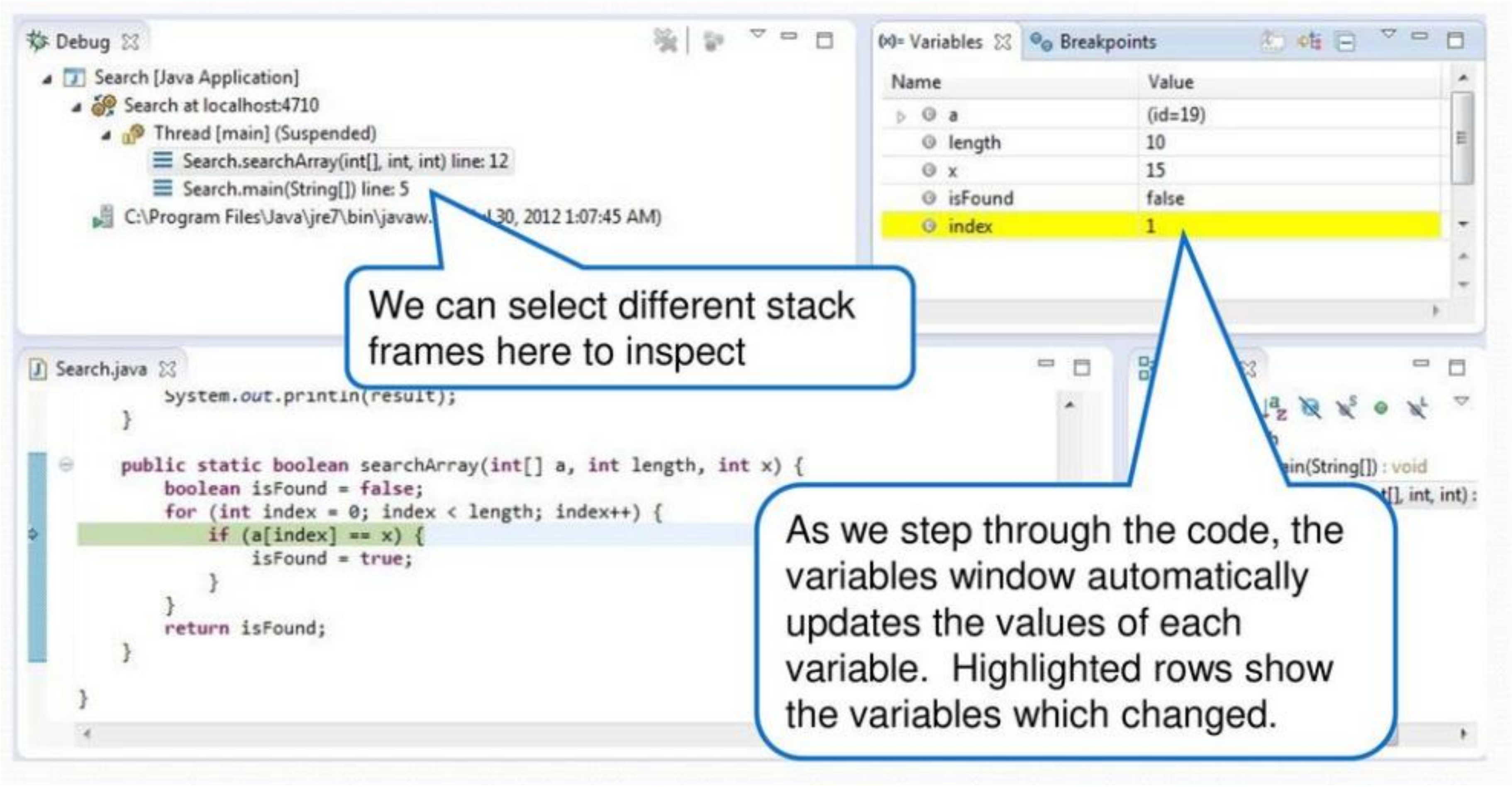
Stepping through code



```
Debug
Search [Java Application]
  Search at localhost:4646
    Thread [main] (Suspended)
      Search.searchArray(int[], int, int) line: 10
      Search.main(String[]) line: 5
  :\Program Files\Java\jre7\bin\javaw.exe (Jul 30, 2012 1:00:58 AM)
```

The program stack frame can help us understand and find out where we are in the current program execution. As calls to methods are made, they are added on top of the ‘stack’. As methods are exited, they are removed from the ‘stack’

From the above example, we can see that we are currently executing the searchArray method, and that it was called by the main method



```
Search [Java Application]
    Search at localhost:4710
        Thread [main] (Suspended)
            Search.searchArray(int[], int, int) line: 12
            Search.main(String[]) line: 5
    C:\Program Files\Java\jre7\bin\javaw.       l 30, 2012 1:07:45 AM)
```

Name	Value
a	(id=19)
length	10
x	15
isFound	false
index	1

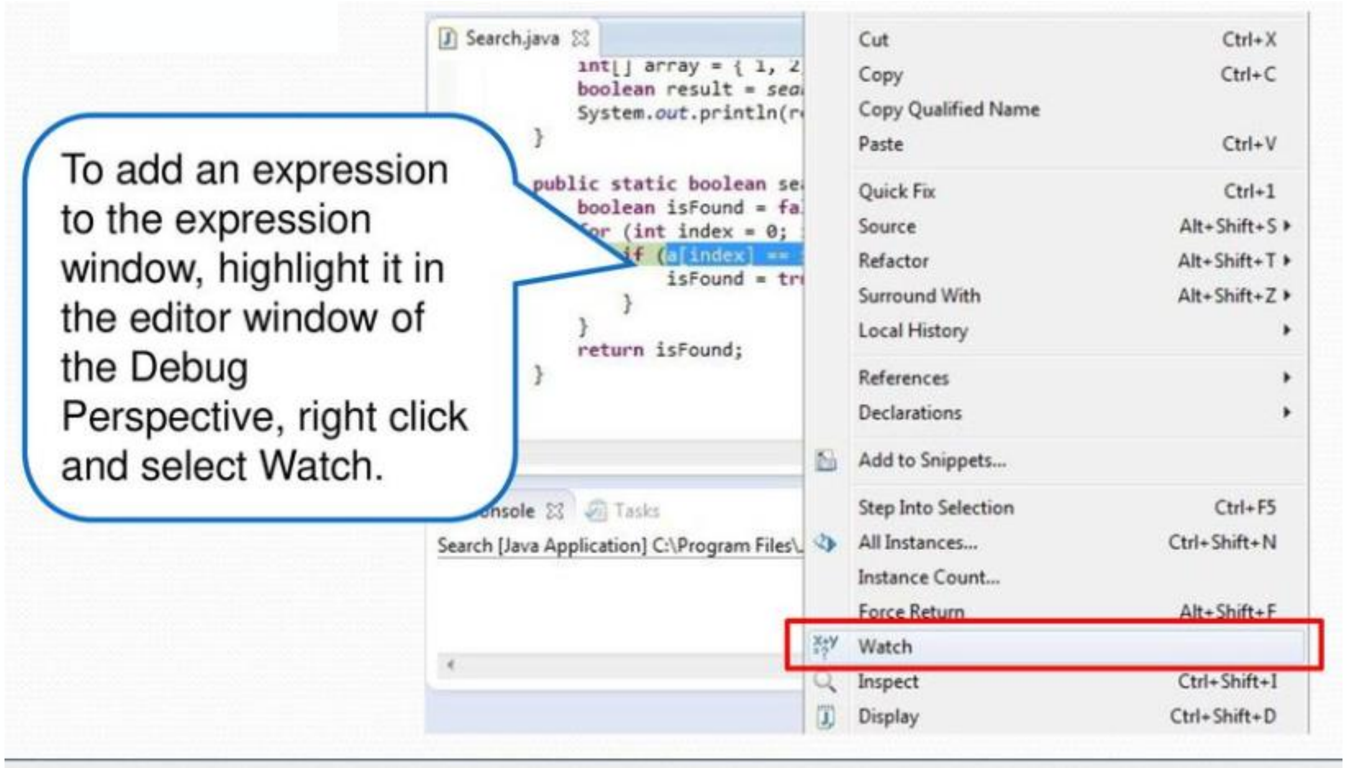
```
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
        }
        return isFound;
    }

}
```

We can select different stack frames here to inspect

As we step through the code, the variables window automatically updates the values of each variable. Highlighted rows show the variables which changed.

Expressions

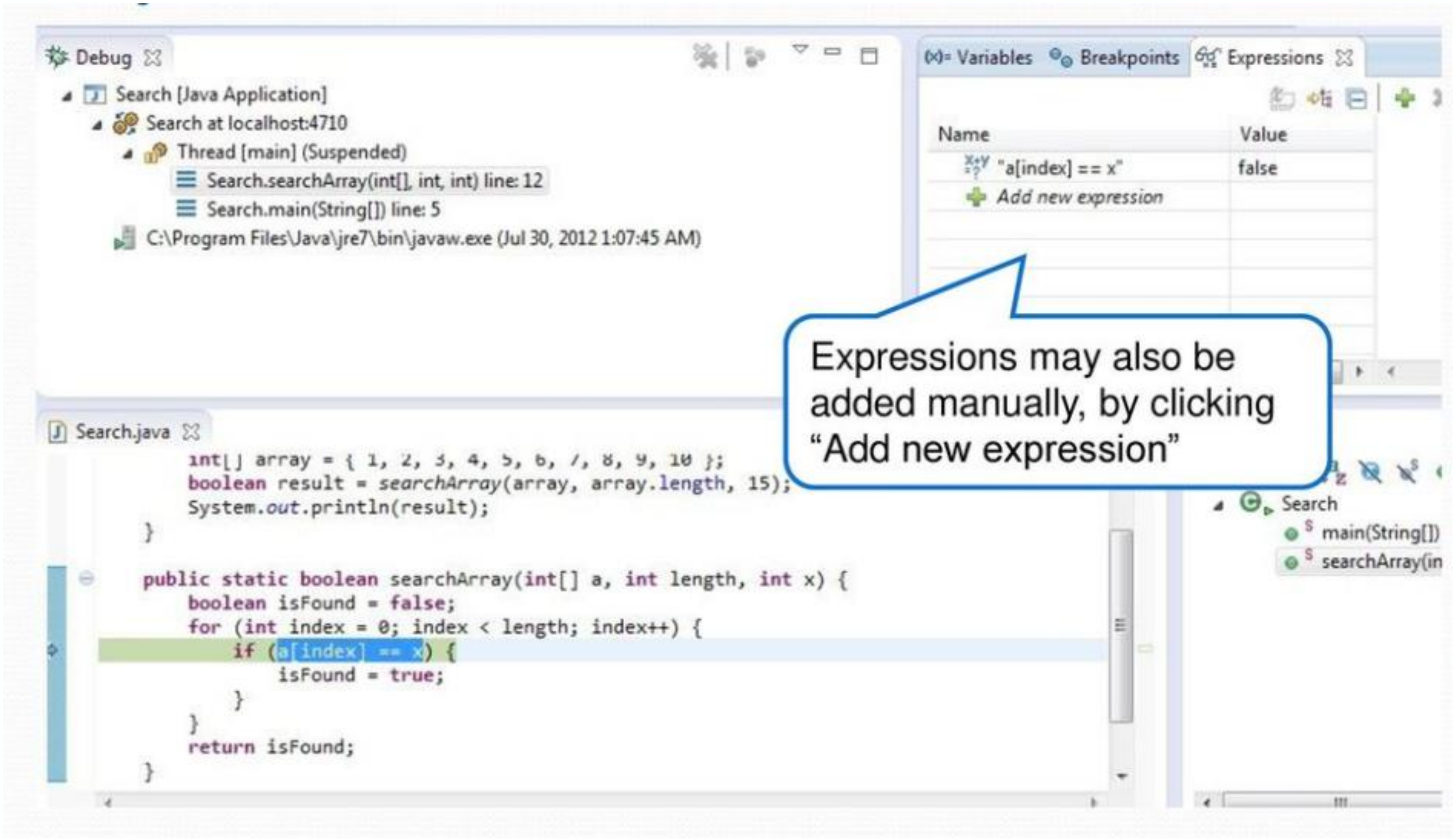
- The variables window automatically keeps track of variables in the current scope of execution
- Eclipse can also keep track of expressions, for example we can add *a[index], or a[index]=x* to the expressions window to examine its value at any time during execution



To add an expression to the expression window, highlight it in the editor window of the Debug Perspective, right click and select Watch.

```
        int[] array = { 1, 2
        boolean result = sea
        System.out.println(r
    }

    public static boolean se
        boolean isFound = fa
        for (int index = 0;
            if (a[index] ==
                isFound = tr
            }
        }
        return isFound;
    }
```

```
Search [Java Application] C:\Program Files\
```



```
Search [Java Application]
  Search at localhost:4710
    Thread [main] (Suspended)
      Search.searchArray(int[], int, int) line: 12
      Search.main(String[]) line: 5
  C:\Program Files\Java\jre7\bin\javaw.exe (Jul 30, 2012 1:07:45 AM)
```

Name	Value
"a[index] == x"	false
Add new expression	

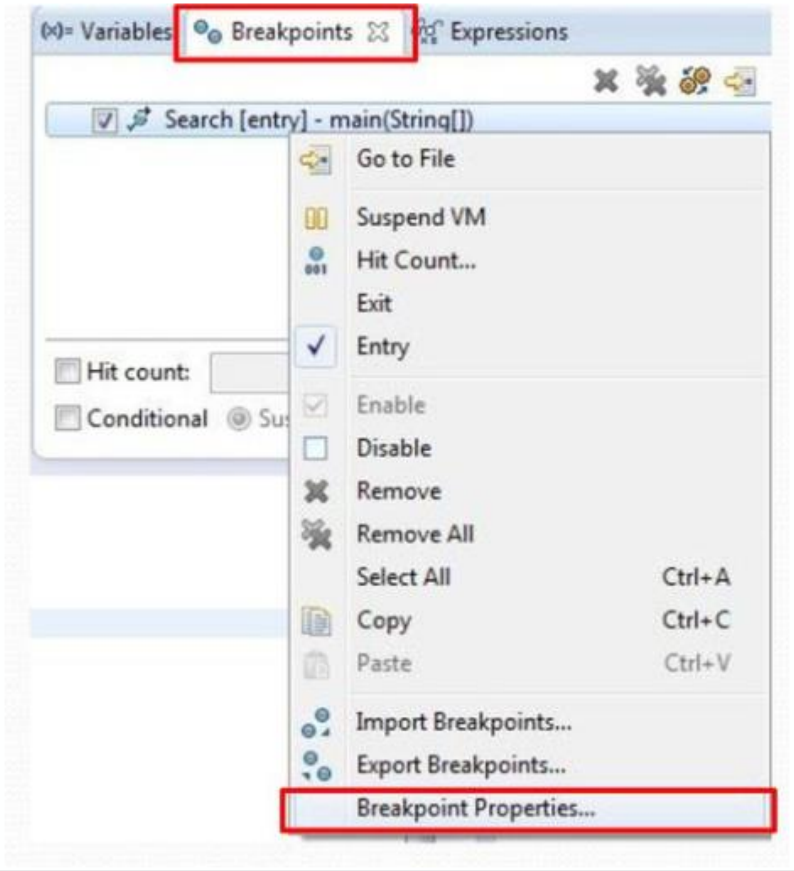
```
        int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        boolean result = searchArray(array, array.length, 15);
        System.out.println(result);
    }

    public static boolean searchArray(int[] a, int length, int x) {
        boolean isFound = false;
        for (int index = 0; index < length; index++) {
            if (a[index] == x) {
                isFound = true;
            }
        }
        return isFound;
    }
```

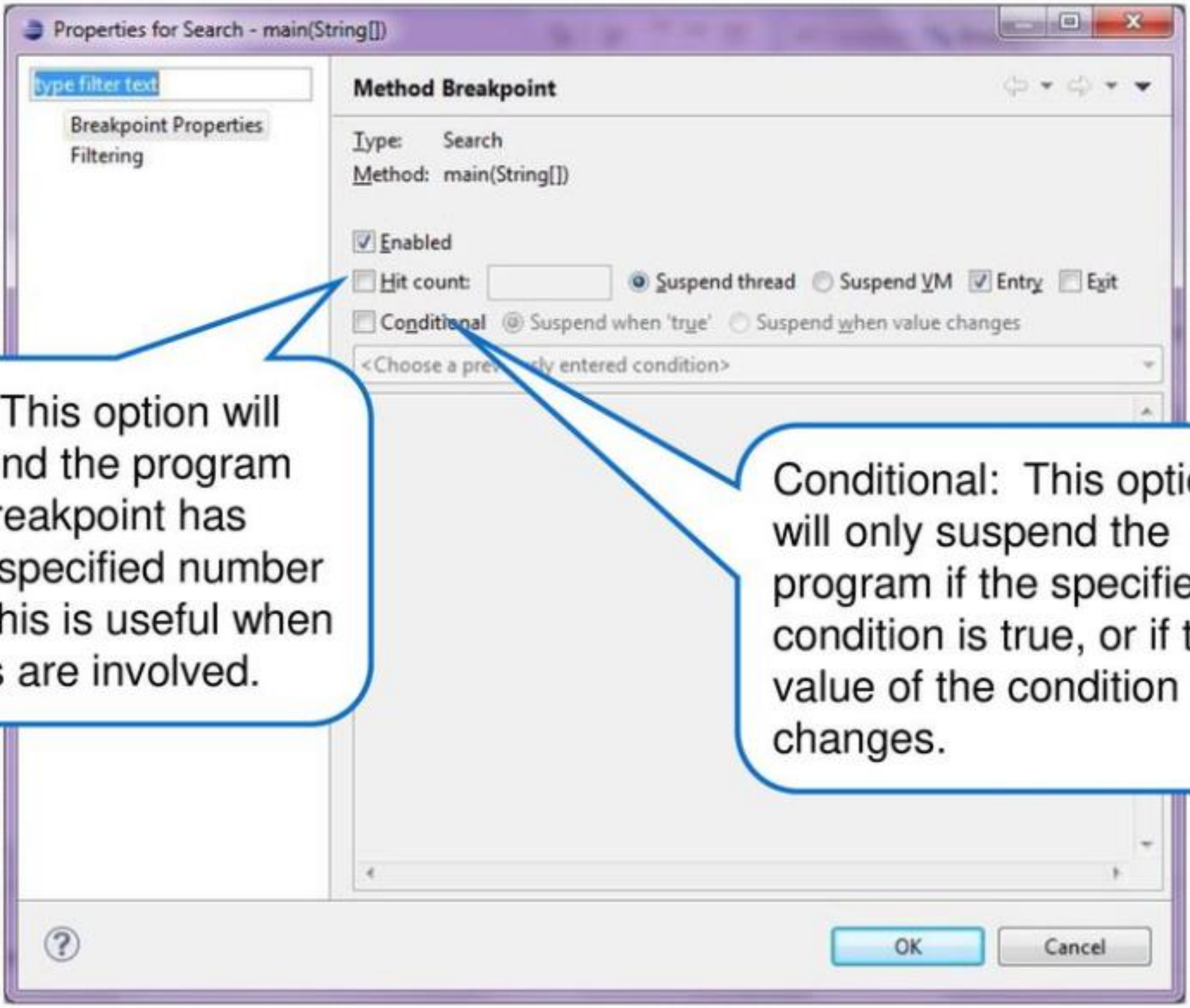
```
Search
  main(String[])
  searchArray(in
```

Expressions may also be added manually, by clicking “Add new expression”

Conditional Breakpoints

- Conditional breakpoints may be used to give the user more flexibility when debugging
- To set conditions on a breakpoint, select the “Breakpoints” tab, right click on a breakpoint, and select “Breakpoint Properties…”





Properties for Search - main(String[])

type filter text

Breakpoint Properties
Filtering

Method Breakpoint

Type: Search
Method: main(String[])

[✓] Enabled
[ ] Hit count: [✓ Suspend thread] Suspend VM [✓] Entry [ ] Exit
[ ] Conditional [✓ Suspend when 'true'] Suspend when value changes
<Choose a previously entered condition>

OK Cancel

Hit count: This option will only suspend the program after the breakpoint has been hit a specified number of times. This is useful when large loops are involved.

Conditional: This option will only suspend the program if the specified condition is true, or if the value of the condition changes.

Summary

https://www.youtube.com/watch?v=9gAjIQc4bPU