Introduction to Programming Paradigms and Core Language Design Issues

Module I

Mrs Pooja Shetty

Department of Information Technology, V.E.S. Institute of Technology, Mumbai.

Why do we need programming language?

Why so many programming language?

Do I need to learn them all?

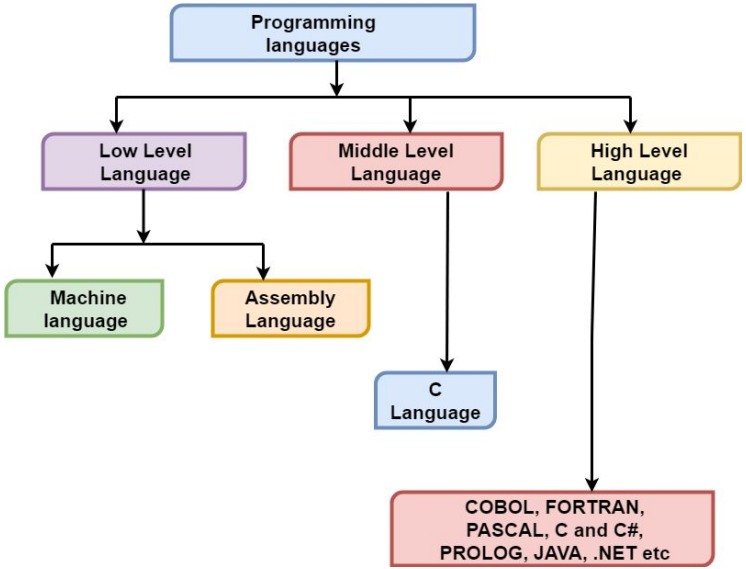
- Evolution - finding better ways to do things
- Special Purposes
- Programmer friendly
- Ease of Implementation
- Support
- Good Compilers/ Interpreters

Why study?

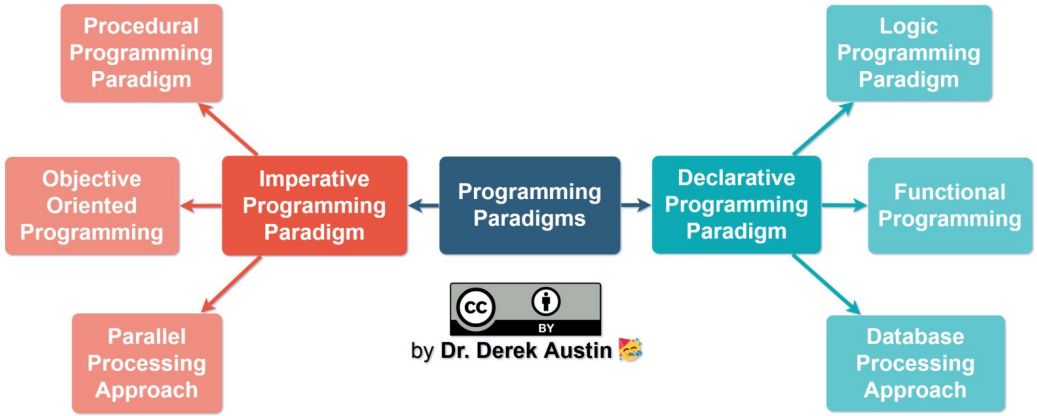
- Choose among alternative ways to express things
- Make good use of debuggers, assemblers, linkers, and related tools

Categories of languages

1)



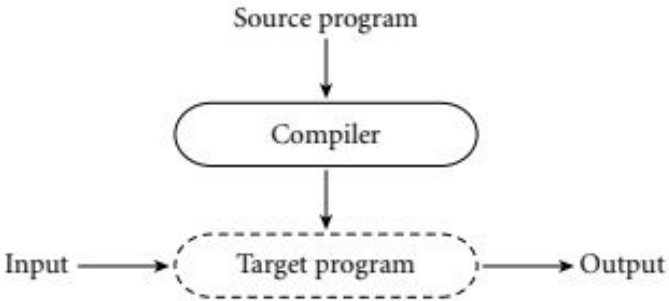
2)

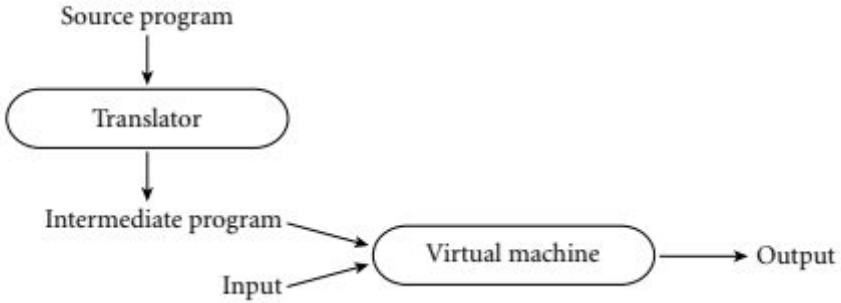


Source :https://www.tutorialandexample.com/low-level-language/

:https://medium.com/javascript-in-plain-english/what-are-javascript-programming-paradigms-3ef0f576dfdb

3) **Compiled and Interpreted Languages**





Compiler	Interpreter
It translates entire program to machine code at once.	It translates single instruction of a program at a time.
It requires more translation time.	It requires less translation time.
Program execution is faster than interpreted languages.	Program execution is slower than compiled languages.
It usually generates additional intermediate code. eg in C (.obj to .exe)	It doesn't generate additional intermediate code. eg in Java (eg: bycode to other format)
It requires more memory as it generates extra object code.	It requires less memory as it does not generate any intermediate code.
Errors are displayed at the end of the compilation process.	Errors are displayed as they met.
Executable code needs to be deployed.	Source code needs to be deployed.
Example of compiled languages – C, C++, etc.	Example of interpreted languages – Ruby, Java Python, Shell script etc.

https://codeforwin.org/2017/05/compiler-and-its-need.html
https://codeforwin.org/2017/05/interpreter-and-its-need.html

What is programming paradignm?

- A **programming paradigm** is a style, or “way,” of programming.
- It’s a concept not a language
- Not a syntax
- It’s about writing / organizing your code.

Source: https://cs.lmu.edu/~ray/notes/paradigms/

- Some of the Common Paradigms
  - **Imperative** (How)
  - **Declarative** (What)
  - **Procedural** (make call to modules)
  - **Functional** (function call, no global var)
  - **Object-Oriented** (Objects and Classes)
  - **Event-Driven** (listens to actions)
  - **Logic** (Rule-based)

Imperative

- Changes the program state through assignment statements.
- Performs step by step task by changing state.
- How to achieve the goal.
- approach to solve problem using some programming language
- after execution of all the result is store

Declarative

- expresses logic of computation without talking about its control flow.
- what needs to be done rather how it should be done

Imperative Programming	Declarative Programming
In this, programs specify how it is to be done.	In this, programs specify what is to be done.
It simply describes the control flow of computation.	It simply expresses the logic of computation.
Its main goal is to describe how to get it or accomplish it.	Its main goal is to describe the desired result without direct dictation on how to get it.
Its advantages include ease to learn and read, the notional model is simple to understand, etc.	Its advantages include effective code, which can be applied by using ways, easy extension, high level of abstraction, etc.
Its type includes procedural programming, object-oriented programming, parallel processing approach.	Its type includes logic programming and functional programming.
In this, the user is allowed to make decisions and commands to the compiler.	In this, a compiler is allowed to make decisions.
It has many side effects and includes mutable variables as compared to declarative programming.	It has no side effects and does not include any mutable variables as compared to imperative programming.
It gives full control to developers that are very important in low-level programming.	It may automate repetitive flow along with simplifying code structure.

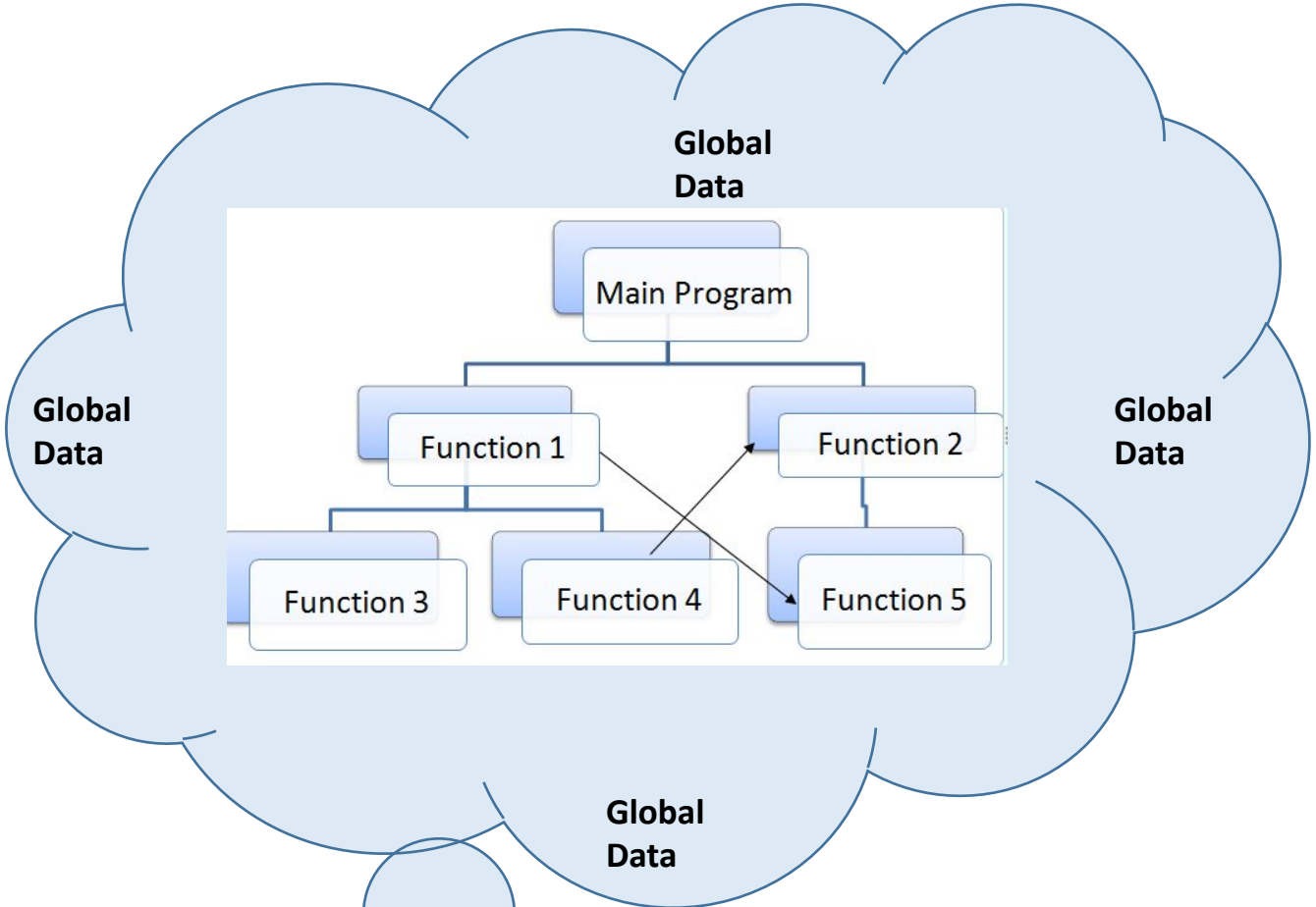
Difference between paradigms

Imperative / **Declarative**

HOW WHAT

```
public class ImperativeVsDeclarativeExample1 {

    public static void main(String[] args) {

        /**
         * Imperative - how style of programming
         */

        int sum=0;
        for(int i=0;i<=100;i++){
            sum+=i;
        }

        System.out.println("Sum using Imperative Approach : " + sum);

        /**
         * Declarative Style of Programming - What style of programming
         */

        int sum1 =IntStream.rangeClosed(0,100) //it splits the values
                .parallel()
                .sum();

        System.out.println("Sum using Declarative Approach : " + sum1);
    }
}
```

Procedural


Global Data
Global Data
Global Data
Global Data
Main Program
Function 1
Function 2
Function 3
Function 4
Function 5

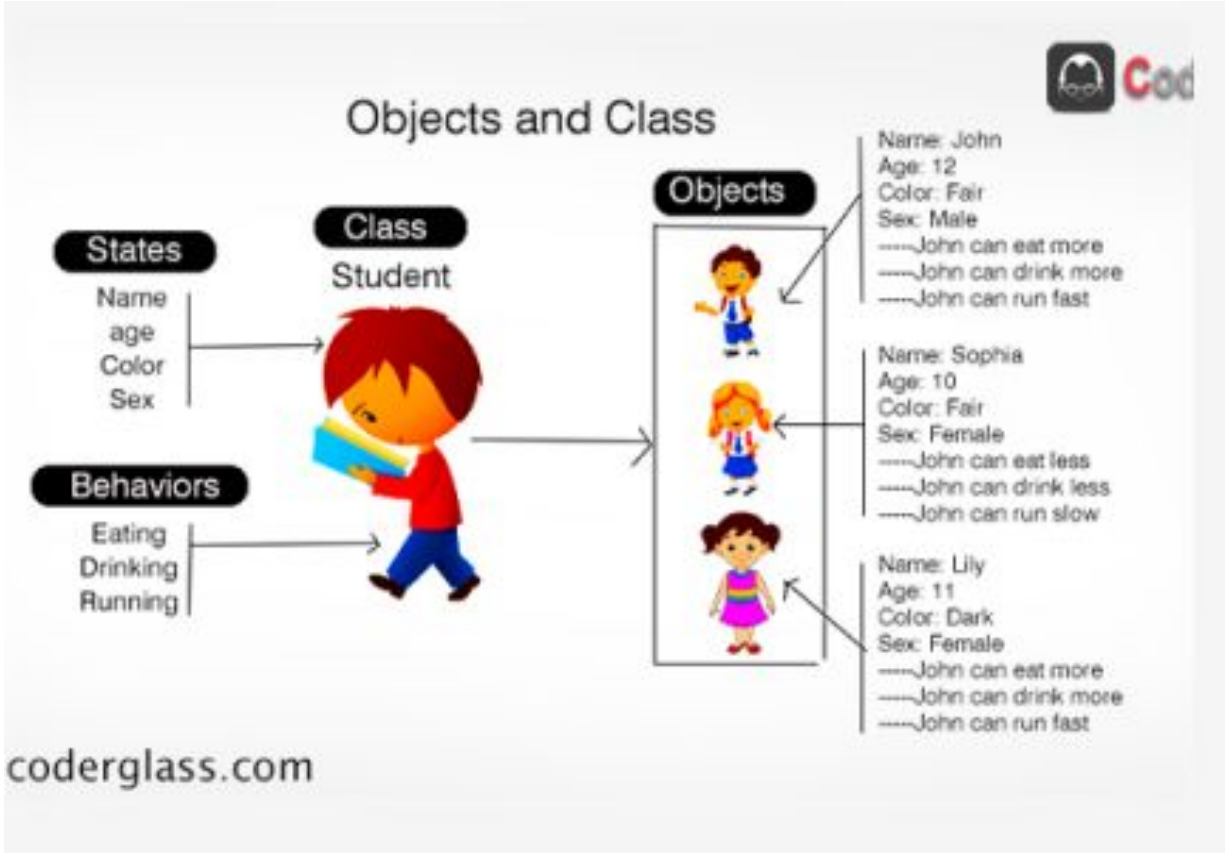
- programming paradigm built around the idea that programs are sequences of instructions to be executed
- Program code is divided up into procedures, discrete blocks of code that carry out a single task.
- Procedures, also known as routines, subroutines, or functions, simply contain a series of computational steps to be carried out in the order specified by the programmer

Procedural

- C

```c
#include<stdio.h>

int main()
{
  int number, Square;

  printf(" \n Please Enter any integer Value : ");
  scanf("%d", &number);

  Square = number * number;

  printf("\n Square of a given number %d is  =  %d", number, Square);

  return 0;
}
```

Object Oriented

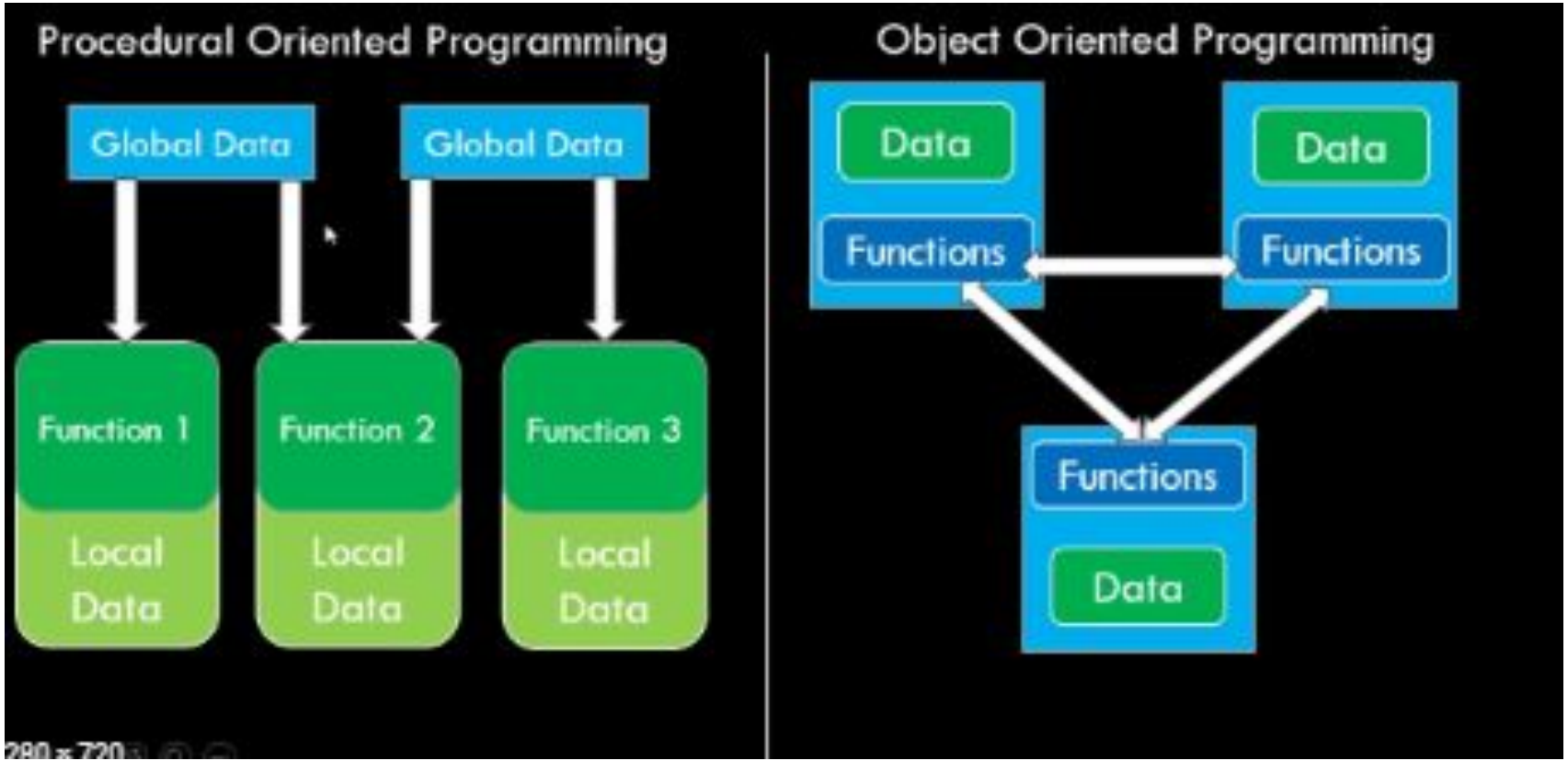


- Emphasis is on data rather procedure.
- Object Oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects.

Object Oriented

- Java

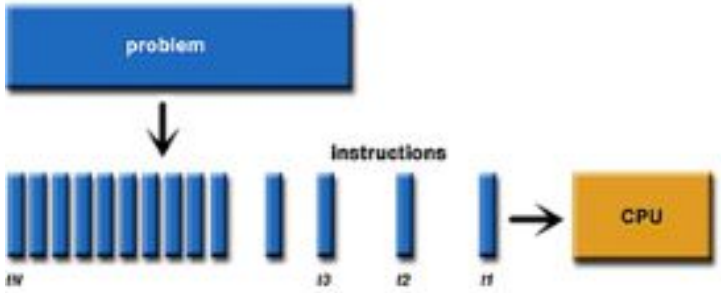
```
import java.util.*;
public class Square
{
public static void main(String args[])
{
Scanner sc=new Scanner(System.in);
int num;
System.out.print("Enter an integer number: ");
num=sc.nextInt();
System.out.println("Square of "+ num + " is: "+ Math.pow(num, 2));
}
}
```

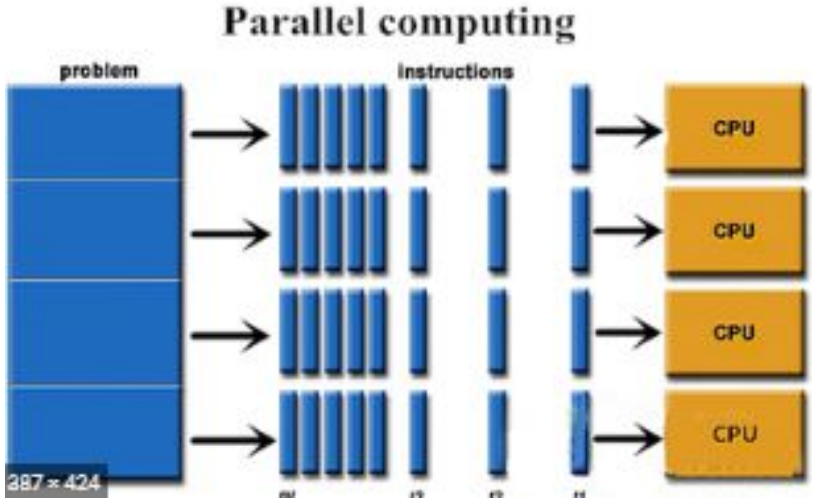


https://www.youtube.com/watch?v=Jt13s8RYA

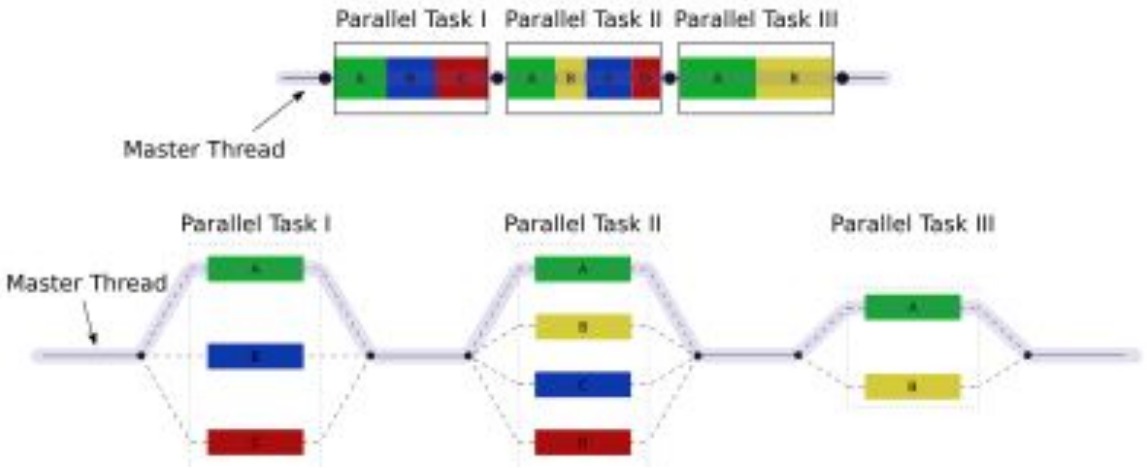
Parallel Processing

- program instructions by dividing them among multiple processors
- Shared memory
- Message passing





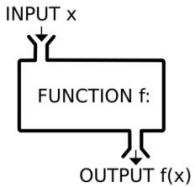
- Java, C#, Go



https://en.wikipedia.org/wiki/Fork%E2%80%93join_model

Functional

- mathematical equation
- immutable data



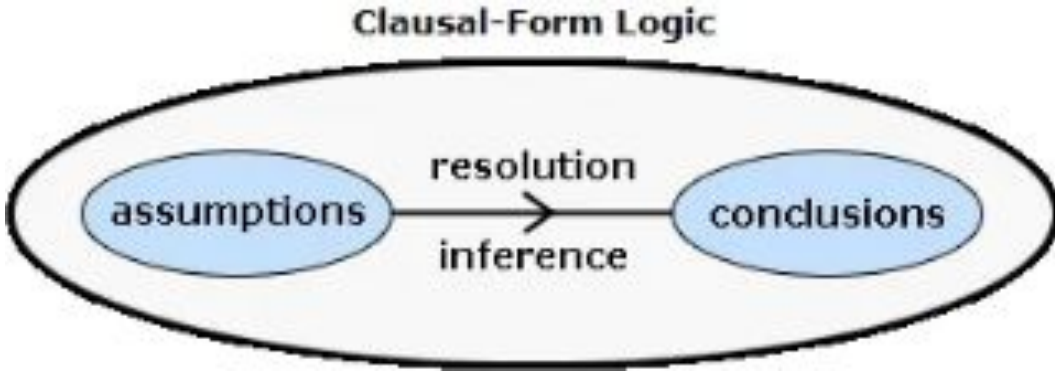
- Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style.
- It uses expressions instead of statements.
- An expression is evaluated to produce a value whereas a statement is executed to assign variables
- Functional Programming is based on Lambda Calculus

- Haskell

```
sq :: Integer -> Integer --function declaration
sq x  =  x * x           --function definition

main = do
   putStrLn "The square of a number is:"
   print(sq 5)       --calling a function
```

Logic

- Logic programming is a computer programming paradigm where program statements express facts and rules about problems within a system of formal logic
- formulas in mathematical logic
- we describe the relationship between the input data and the output data
- the computer figure out how to obtain the output from the input



- Prolog

```
square(N1):-
    Ans is N1 * N1,
write(Ans).
```

Data driven

- model primitive functional nodes.
- They provide an inherently parallel model:
- nodes are triggered by the arrival of input tokens

- SQL

```
Select EmployeeName,Gender,Salary
 from Employee
 ORDER BY  CASE Gender
WHEN 'F' THEN Salary End DESC,
Case WHEN Gender='M' THEN Salary
END
```

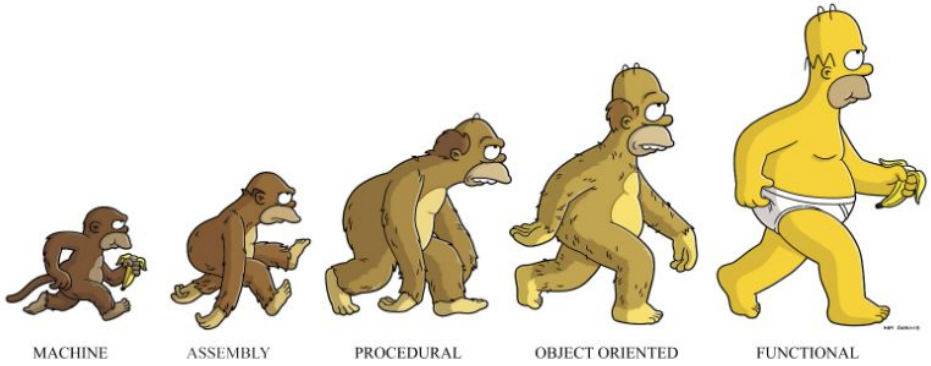
Ascending Order

00 %

Results | Messages

	EmployeeName	Gender	Salary
1	Stella	F	76000.00
2	Jorge	F	75000.00
3	Nicholas	F	71000.00
4	Ernest	F	64000.00
5	Gilbert	M	42000.00
6	Salvador	M	75000.00
7	Jerome	M	83000.00
8	Ray	M	88000.00
9	Edward	M	93000.00
10	Lawrence	M	95000.00

- Smalltalk –Object oriented
- Haskel –Fucntional
- Python - procedural, object oriented, imperative, functional
- C++, PHP – Object oriented / Procedural
- CSS - Declarative



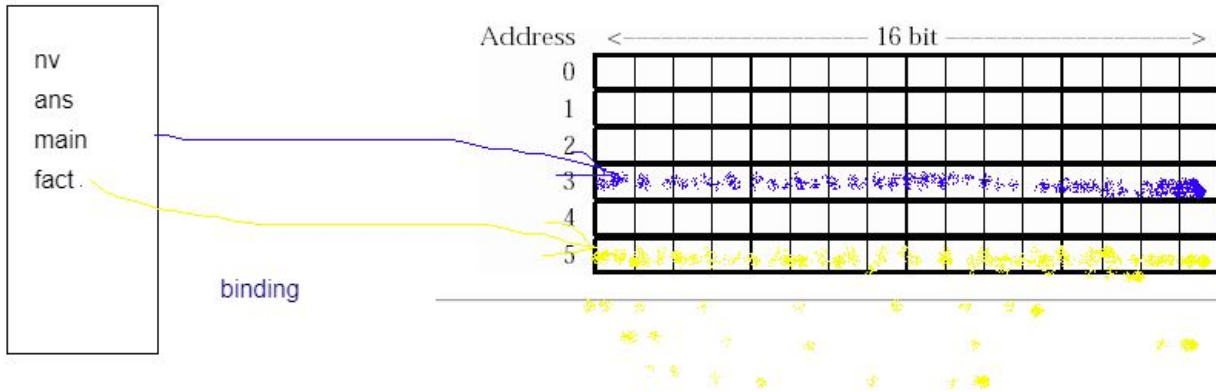
Core Language Design Issues

- High level programming languages introduced **high level of abstraction**
- By abstracting the language away from the hardware, designers not only made it possible to write programs that would run well on a wide variety of machines, but also made the programs easier for human beings to understand
- By hiding irrelevant details, abstraction reduces conceptual complexity, making it possible for the programmer to focus on a manageable subset of the program text at any particular time

Name

- A name is a mnemonic character string used to represent something else.
- Names in most languages are identifiers
- Identifiers like variables, constants, types and so on using symbolic identifiers rather than low-level concepts like addresses
- Names introduce data abstraction and control abstraction
- These names need to be binded to actual memory locations (addresses)

```
#include <iostream>
using namespace std;
int fact(int n); // function prototype

int main(){
int nv,ans;
      cout<<" Input n:";
      cin>> nv;
      ans = fact(nv);// Function is called
      cout<<"Answer is:"<<ans<<endl;
}

int fact(int n){  // This function maps input int to another int
      int ans =1; // ans is the local variable of the function
      for(int i= 2; i<= n; i++)     // i also a local sractch pad variable
          ans *=i;
      return ans; // function returns the answer to the caller
}
```



Binding

Binding

- A binding is an association between two things
  - Name of a variable and its address
  - Name of the operator and its meaning

- Binding time is the time at which a binding is created or, more generally,the time at which any implementation decision is made
  - Which name to which address?

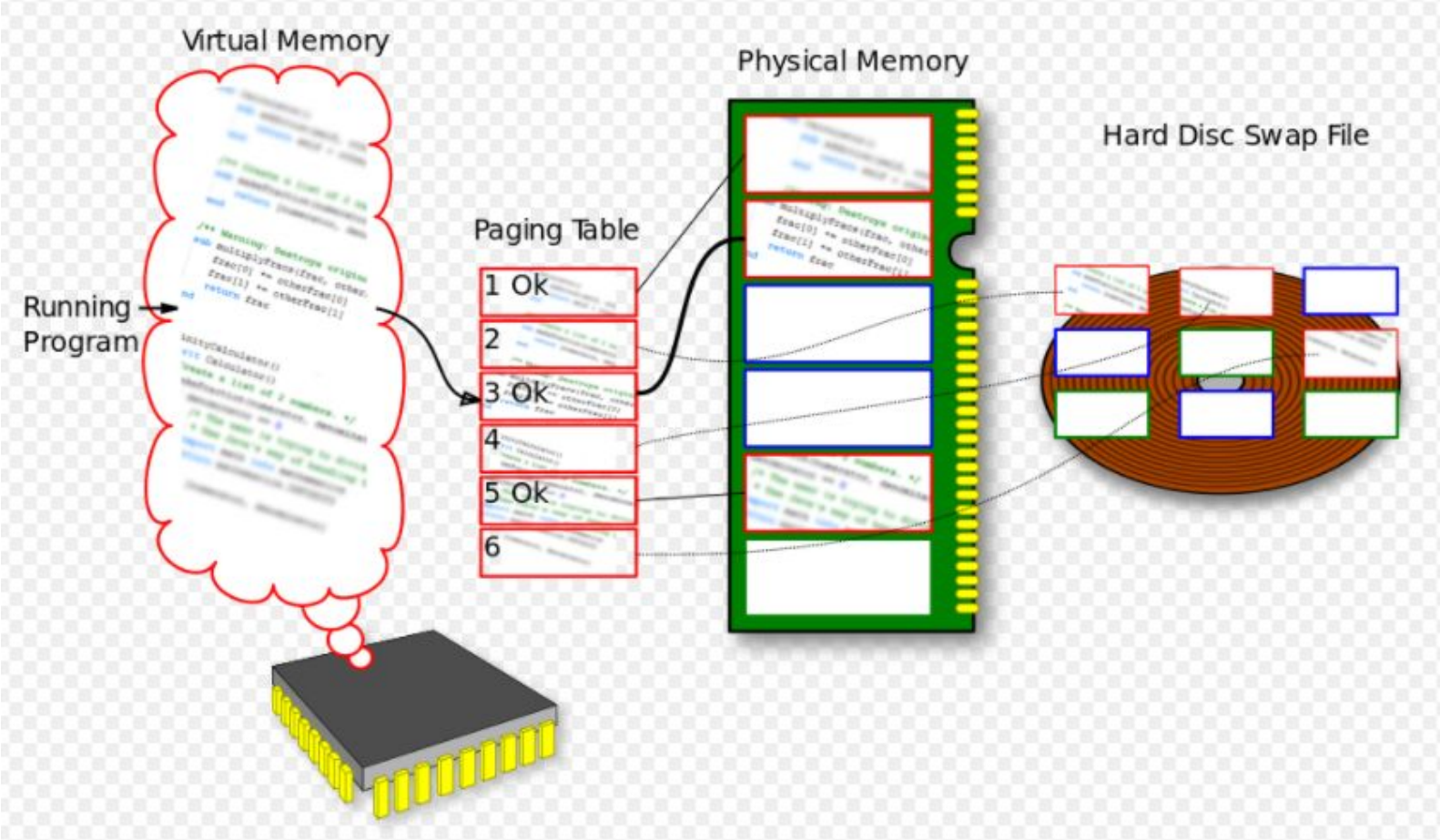
- Broadest way of classifying Binding time is
  - Runtime /*Late binding*
  - Compile time / *Early binding*

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	Overloaded function call, overloaded operators.	Virtual function in C++, overridden methods in java.

Different times at which decisions may be bound:

- **Language design time:** Aspects of language semantics are chosen when the language is designed.
  - **control flow** constructs
  - primitive types constructors for creating complex types eg: **keywords**)
- **Language implementation time:** Most language manuals leave a variety of issues to the discretion of the language implementor.
  - precision (number of bits) of the fundamental types
  - I/O to the operating system's notion of files
  - the organization and maximum sizes of stack and heap, and the handling of run-time
  - exceptions such as arithmetic overflow.
  - max identifier name length

- **Program writing time:** Programmers, of course, choose algorithms, data structures and names.
- **Compile time:** Compilers choose the mapping of high-level constructs to machine code, including the layout of statically defined data in memory.
- **Link time:** Since most compilers support separate compilation compiling different modules of a program at different times and depend on the availability of a library of standard subroutines
  - a program is usually not complete until the various modules are joined together by a linker.
  - Virtual memory mapping
  - When a name in one module refers to an object in another module, the binding between the two is not finalized until link time.
  - External symbols are linked to a specific set of object files and libraries



https://en.wikipedia.org/wiki/Memory_address#/media/File:Paging.svg

- **Load time:** Load time refers to the point at which the operating system loads the program into memory so that it can run. In primitive operating systems,
  - Most modern operating systems distinguish between virtual and physical addresses. Virtual addresses are chosen at link time;
  - Physical addresses can actually change at run time.
  - The processor's memory management hardware translates virtual addresses into physical addresses during each individual instruction at run time.
- **Run time**: Run time is actually a very broad term that covers the entire span from the beginning to the end of execution
  - Bindings of values to variables occur at run time,

Scope and Scope Rules

Scope

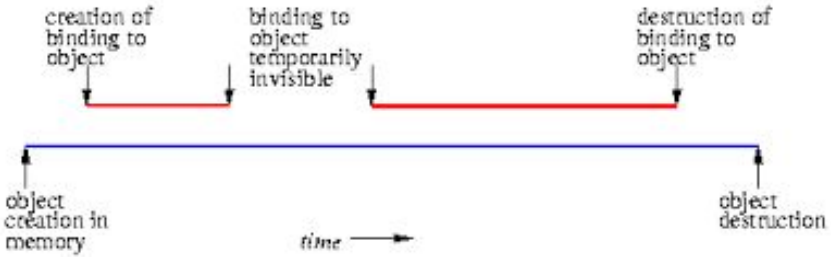
- The textual region of the program in which a binding is active is its **scope**
- The complete set of bindings in effect at a given point in a program is known as the current **referencing environment**

Lifetime and Storage Management

Bindings key events:

- creation of objects
- creation of bindings
- references to variables (which use bindings)
- (temporary) deactivation of bindings
- reactivation of bindings
- destruction of bindings
- destruction of objects

- The period of time between the creation and the destruction of a name-to- object binding is called the binding's lifetime.
- The time between the creation and destruction of an object is the **object's lifetime.**



- If **object outlives** binding it's garbage

- If **binding outlives object** it's a dangling reference, e.g., if an object created via the C++ new operator is passed as a & parameter and then deallocated (delete-ed) before the subroutine returns

Scope Rules

Three rules

The scope of an entity is the program or function in which it is declared.

```
int main()
        {
                int i=4;
                cout<<i;
        }
```

A global entity is visible to all contained functions, including the function in which that entity is declared.

```
int i =1;
int main()
{
        cout<<i;
}
```

An entity declared in the scope of another entity is always a different entity even if their names are identical.

```
int main()
{
        int i=4;
        for(int i=1;i<10;i++);
        cout<< i;
}
```

Storage Management

Storage Allocation mechanisms are used to manage the object's space:

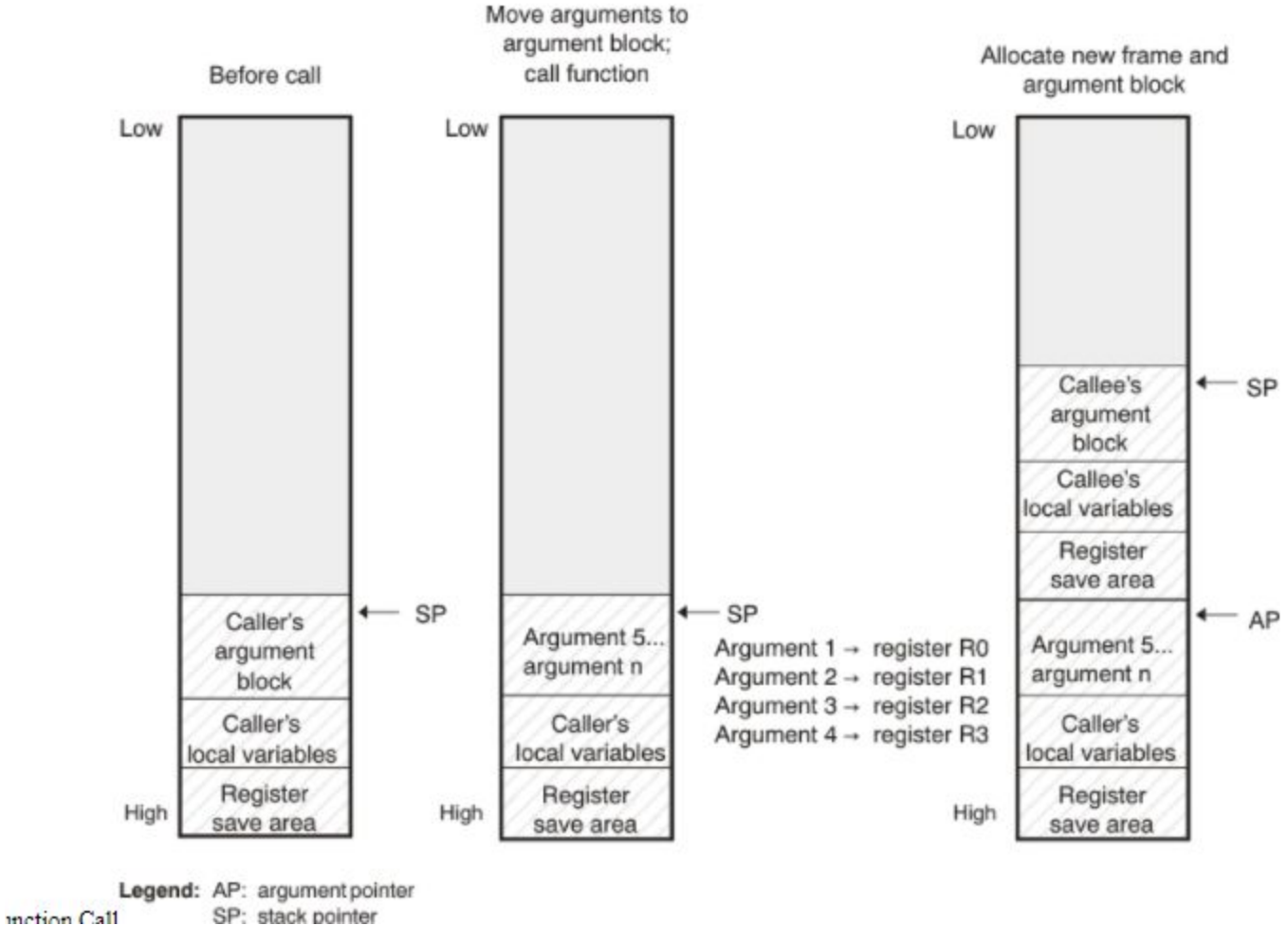
- **Static**: the objects are given an absolute address that is retained throughout the program's execution
- **Stack**: the objects are allocated and deallocated in last-in, first-out order, usually in conjunction with subroutine calls and returns.
- **Heap**: the objects may be allocated and deallocated at arbitrary times (require a complex storage management mechanism).

Static allocation for:

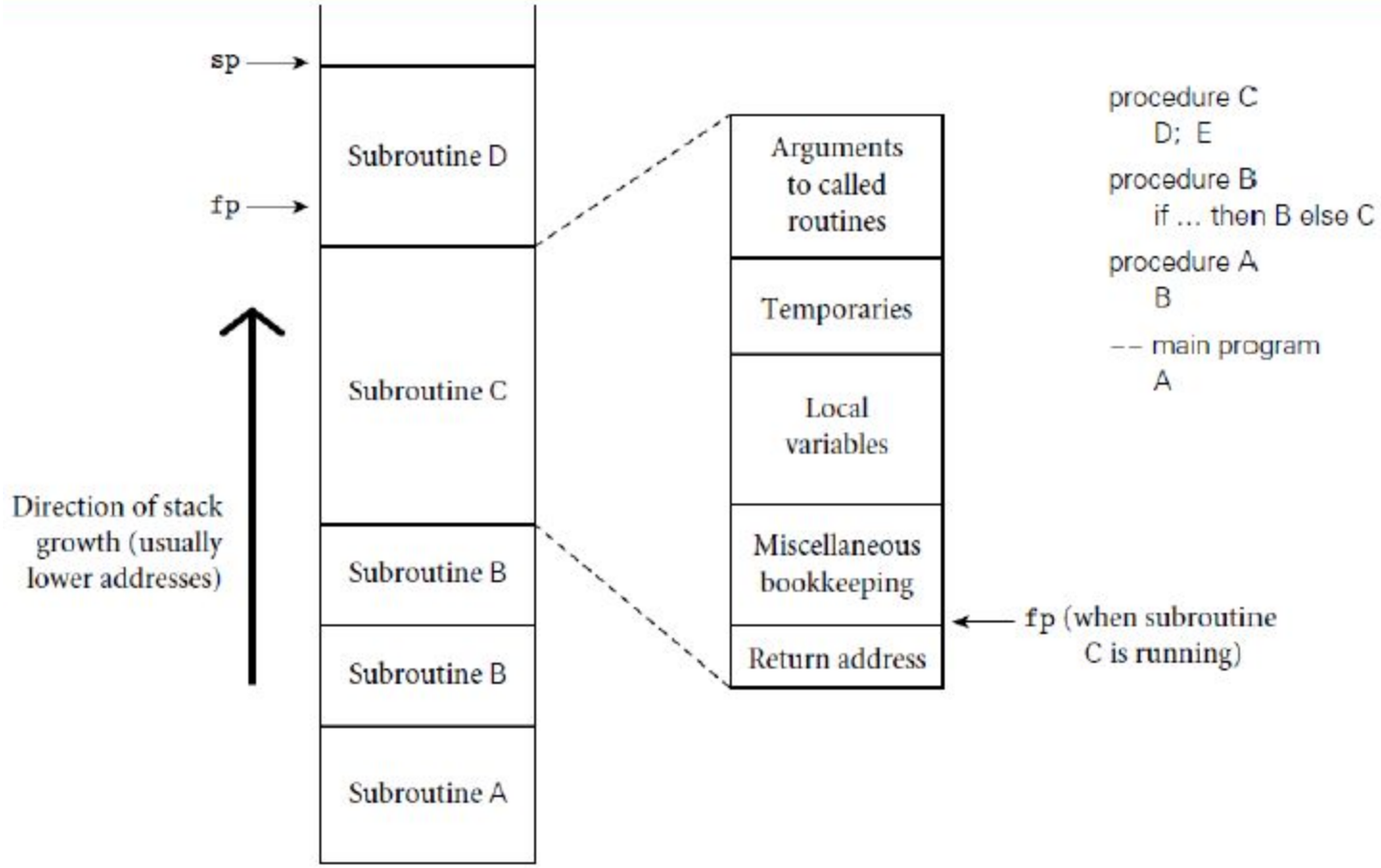
- code (and small constants often stored within the instruction itself)
- globals
- static or own variables
- explicit constants (including strings, sets, etc.), e.g., printf ("hello, world n")
- Arguments and return values
- Temporaries (intermediate values produced in complex claculations)
- Bookkeeping information (subroutine’s return address, a reference to the stack frame of the caller (the dynamic link), additional saved registers, debugging information

Stack

- Why a stack
  - allocate space for recursive routines
  - reuse space
- Each instance of a subroutine at run time has its own frame (or activation record) for
  - parameters
  - local variables
  - temporaries (return address)
- Maintenance of the stack is the responsibility of the subroutine calling sequence (the code executed by the caller immediately before and after the call),which includes: the ***prologue*** (code executed at the beginning) and ***epilogue*** (code executed at the end) of the subroutine itself.



http://downloads.ti.com/docs/esd/SPNU151T/function-structure-and-calling-conventions-stdz0545494.html

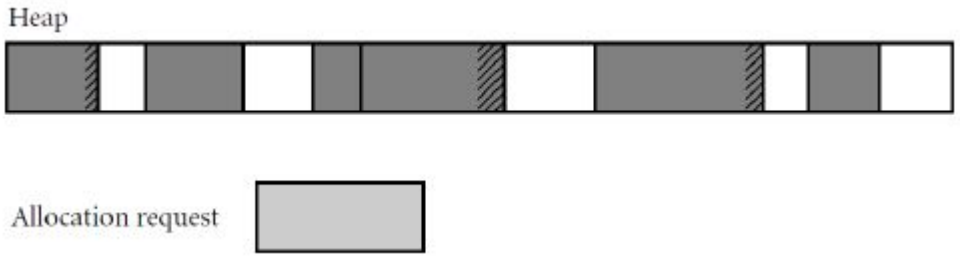

sp
fp
Subroutine D
Subroutine C
Subroutine B
Subroutine B
Subroutine A
Direction of stack growth (usually lower addresses)
Arguments to called routines
Temporaries
Local variables
Miscellaneous bookkeeping
Return address
fp (when subroutine C is running)
procedure C
D; E
procedure B
if ... then B else C
procedure A
B
-- main program
A

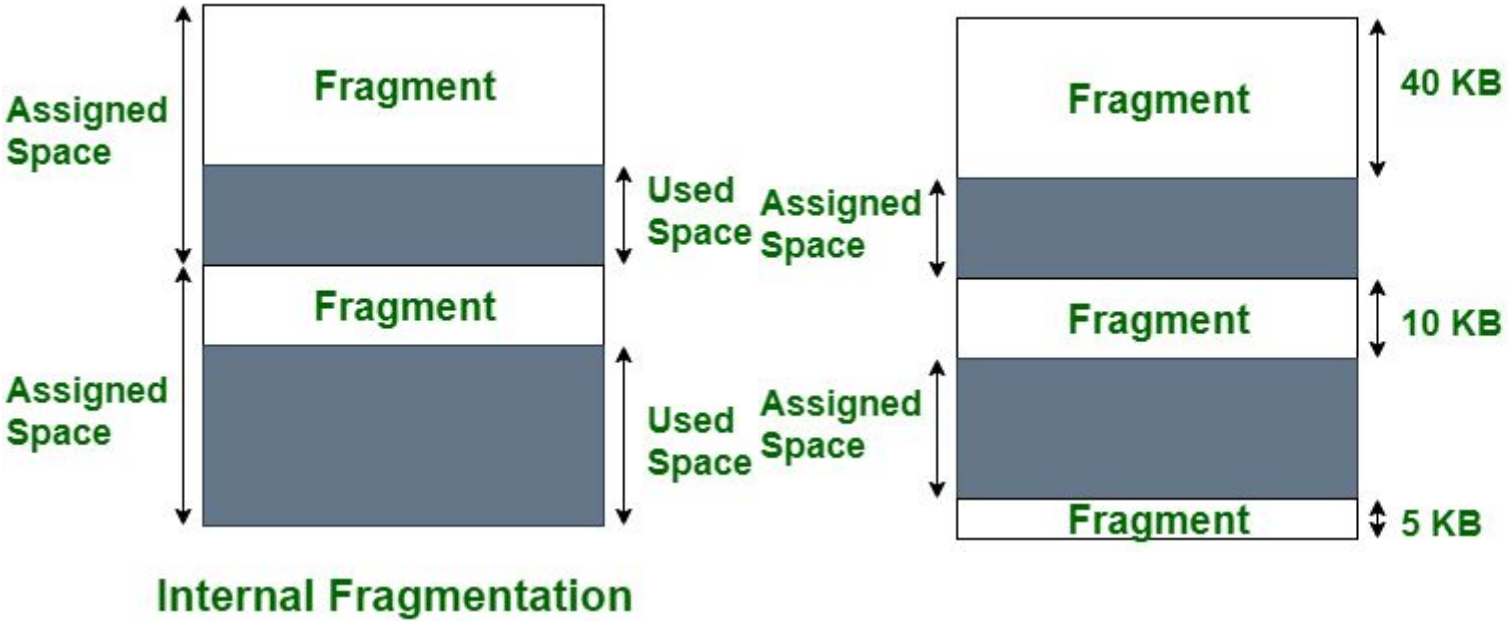
Heap Based Allocation

- Heap is for dynamic allocation
- A heap is a region of storage in which sub blocks can be allocated and deallocated at arbitrary times
    - dynamically allocated pieces of data structures:
      objects, Strings, lists, and sets, whose size may change as a result of an assignment statement or other update operation

Fragmentation :

- **Internal fragmentation** occurs when a storage management algorithm allocates a block that is larger than required to hold a given object.
- **External fragmentation** occurs when the blocks that have been assigned to active objects are scattered through the heap in such a way that the remaining, unused space is composed of multiple blocks: there may be quite a lot of free space, but no one piece of it may be large enough to satisfy some request
    - compaction, paging or segmentation





Internal Fragmentation

Process 07
needs 50K
memory sp

S.NO	INTERNAL FRAGMENTATION	EXTERNAL FRAGMENTATION
1.	In internal fragmentation fixed-sized memory, blocks square measure appointed to process.	In external fragmentation, variable-sized memory blocks square measure appointed to method.
2.	Internal fragmentation happens when the method or process is larger than the memory.	External fragmentation happens when the method or process is removed.
3.	The solution of internal fragmentation is best-fit block.	Solution of external fragmentation is compaction, paging and segmentation.
4.	Internal fragmentation occurs when memory is divided into fixed sized partitions.	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
5.	The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, is called External fragmentation .

- The storage management algorithm maintains a single linked list, the free list , of heap blocks not currently in use

- The **first fit algorithm** selects the first block on the list that is large enough to satisfy a request
- The **best fit algorithm** searches the entire list to find the smallest block that is large enough to satisfy the request
- Common mechanisms for dynamic pool adjustment:
    - The buddy system : the standard block sizes are powers of two
    - The Fibonacci heap: the standard block sizes are the Fibonacci numbers
    - Compacting the heap moves already allocated blocks to free large blocks of space

Garbage Collection

- In languages that deallocation of objects is not explicit
  - Manual deallocation errors are among the most common and costly bugs in real world programs
- Objects are to be deallocated implicitly when it is no longer possible to reach them from any program variable
  - Costly

Static Scoping

-

Type Systems

Type Systems

- bits in memory can be interpreted in different ways
  – instructions
  – addresses
  – characters
  – integers, floats, etc.
- bits themselves are untyped, but high-level languages associate types with values
  – to provide the contextual information
  – allows error checking

Type Systems, Type Checking, Equality Testing and Assignment.

Definition: A type system is a set of types and type constructors (arrays, classes, etc., built from existing primitive type) along with the rules that govern whether or not a program is legal with respect to types (i.e., type checking).

Definition: A type constructor is a mechanism for creating a new type

Eg: ARRAY is a type constructor since when we apply it to an index type and an element type it creates a new type: an array type. class in C++ is another example of a type constructor.

As an extreme example, one can do this in C++ but not in Java: int x = (int) “Hello”;

Why do different languages use different type systems?

- no one perfect type system.
- Each type system has its strengths and weaknesses. Thus, different languages use different type systems because they have different priorities.
- A language designed for writing operating systems is not appropriate for programming the web; thus they will use different type systems. When designing a type system for a language, the language designer needs to balance the tradeoffs between execution efficiency, expressiveness, safety, simplicity, etc.

Thus, a good understanding of type systems is crucial for understanding how to best exploit programming languages.

- a type system consists of
  – a way to define types and associate them with language constructs
    - constructs that must have values include constants, variables, record fields, parameters, literal constants, subroutines, and complex expressions containing these
  – rules for type equivalence, type compatibility, and type inference
    - **type equivalence**: when the types of two values are the same
    - **type compatibility**: when a value of a given type can be used in a particular context
    - **type inference**: type of an expression based on the types of its part,
      -

Type Checking

- type checking ensures a program obeys the language's type compatibility rules
  - – type clash: violation of type rules
- Strongly/ Weakly
  - a language is ***strongly typed*** if it prohibits an operation from performing on an object it does not support
    - – A strongly-typed language is one in which variables are bound to specific data types, and will result in type errors if types do not match up as expected in the expression
    - – eg : Java, Python
  - Weakly-typed languages make conversions between unrelated types implicitly

- Statically/ Dynamically
  - a language is ***statically typed*** if it is strongly typed and type checking can be performed at compile time
    – eg :Java, C, C++, Haskell, FORTRAN, Pascal and Scala

  - a language is ***dynamically typed*** if type checking is performed at run time
    – late binding
    – List and Smalltalk
    – scripting languages, such as Python and Ruby

Polymorphism

- polymorphism results when the compiler finds that it doesn't need to know certain things
  – allows a single body of code to work with objects of multiple types
  – with dynamic typing, arbitrary operations can be applied to arbitrary objects
  - implicit parametric polymorphism: types can be thought of to be implied unspecified parameters
  - incurs significant run-time costs
  - Eg: ML’s type inference/unification scheme

- subtype polymorphism in object-oriented languages
  – a variable of the base type can refer to an object of the derived type
  – explicit parametric polymorphism, or generics
    - C++, Eiffel, Java
    - useful for container classes List where T is left unspecified

Data Types

- we all have developed an intuitive notion of what types are that they are of 3 kinds

  – collection of values from a domain (the ***denotational*** approach)
  - approach for providing mathematical meaning to systems and programming languages by constructing mathematical objects (called denotations) that describe the meanings of expressions from the languages.

– internal structure of data, described down to the level of a small set of fundamental types (the ***structural*** approach)

- type is either one of a small collection of built-in types (integer, character, Boolean, real, etc.; also called primitive or predefined types),

  or

- a composite type created by applying a type constructor (record, array, set, etc.)to one or more simpler types

– collection of well-defined operations that can be applied to objects of that type (the ***abstraction*** approach)

Classification of Types

- most languages provide a set of built-in types
  - integer
  - boolean
    - often single character with 1 as true and 0 as false
    - C: no explicit boolean; 0 is false, anything else is true
  - char
    - traditionally one byte
    - ASCII, Unicode
  - real

- numeric types
  – C and Fortran distinguish between different lengths of integers
  – C, C++, C#, Modula-2: signed and unsigned integers
  – differences in real precision cause unwanted behavior across machine platforms
  – some languages provide complex, rational, or decimal types

- enumeration types
  – An enumeration type consists of a set of named elements
  – facilitate readability
  – allow compiler to catch errors
  – An enumeration type consists of a set of named elements
    - Pascal
      type weekday = (sun, mon, tue, wed, thu, fri, sat);
      for today := mon to fri do begin …

- The ordered nature of enumerations facilitates the writing of enumeration-controlled loops

  for today := mon to fri do begin …

- It also allows enumerations to be used to index arrays:

  var daily_attendance : array [weekday] of integer;

- C

  enum weekday {sun, mon, tue, wed, thu, fri, sat};

  or

  typedef int weekday;
  const weekday sun = 0, mon = 1, tue = 2,
  wed = 3, thu = 4, fri = 5, sat = 6;

- subrange types
  – contiguous subset of discrete base type
  – helps to document code
  – most store the actual values rather than ordinal locations
    - Pascal
      type test_score = 0..100;
      workday = mon..fri;
    - Ada
      type test_score is new integer range 0..100;
      subtype workday is weekday range mon..fri;

- composite types
  - – records (structures)
    - Cobol
    - fields of possibly different type
    - similar to tuples
  - – variant records (unions)
    - multiple field types, but only one valid at a given time
  - – arrays
    - as a function that maps members of an index type to members of a component type.
  - – sets
    - Discrete
    - like enumerations and subranges

– pointers
    - reference to an object of the pointer’s base type.
    - good for recursive data types
– lists
    - head element and following elements
    - variable length, no indexing
– files
    - mass storage, outside program memory
    - linked to physical devices (e.g., sequential access)

Orthogonality

– operations change just one thing without affecting others.

– C and Void (empty) type Algol 68, for example, a subroutine that is meant to be used as a procedure is generally declared with a “return” type of void.

Type Checking

- a type system has rules for
  – ***type equivalence*** (when are the types of two values the same?)
  – ***type compatibility*** (when can a value of type A be used in a context that expects type B?)
    - object can be used if its type and the type expected by the context are equivalent
  – ***type inference*** (what is the type of an expression, given the types of the operands?)
    - Whenever an expression is constructed from simpler subexpressionswhat is the type of the expression as a whole?

Type Equivalence

- two major approaches:
  - structural equivalence
  - name equivalence

- ***Structural equivalence*** is based on the content of type definitions: roughly speaking, two types are the same if they consist of the same components, put together in the same way.

  – structural equivalence is based on some notion of meaning behind those declarations

  – Algol-68, Modula-3 C and ML.

- format does not matter
  struct { int a, b; }
  is the same as
  struct { int b, a; }
  want them to be the same as
  struct { int a; int b; }

In Pascal

```
type R2 = record
    a, b : integer
end;
```

should probably be considered the same as

```
type R3 = record
    a : integer;
    b : integer
end;
```

But what about

```
type R4 = record
    b : integer;
    a : integer
end;
```

```
1.  type student = record
2.          name, address : string
3.          age : integer

4.  type school = record
5.          name, address : string
6.          age : integer

7.  x : student;
8.  y : school;
9.  ...
10. x := y;                 -- is this an error?
```

- programmers would probably want to be informed if they accidentally assigned a value of type school into a variable of type student,
- but a compiler whose type checking is based on structural equivalence will accept such an assignment.

- Name equivalence is based on the lexical occurrence of type definitions: roughly speaking, each definition introduces a new type. based on the assumption that if the programmer goes to the effort of writing two type definitions, then those definitions are probably meant to represent different types

  – name equivalence is based on declarations

  – Java, C#, standard Pascal,

  – Type Conversion and Casts

  – NonconvertingType Casts

Type Compatibility

- coercion
  – when an expression of one type is used in a context where a different type is expected, one normally gets a type error
  – var a : integer; b, c : real; ... c := a + b;
  – many languages allow such statements, and coerce an expression to be of the proper type
  – coercion can be based just on types of operands, or can take into account expected type from surrounding context as well
  – Fortran has lots of coercion, all based on operand type
  – C has lots of coercion, too, but with simpler rules:
    – all floats in expressions become doubles
    – short int and char become int in expressions
    – if necessary, precision is removed when assigning into LHS

– Overloading and Coercion

- Some operators such as + are overloaded: + has several possible types for example:
  - int +(int,int),
  - float +(float, float)
  - but also float* +(float*, int)
  - int* +(int, int*) depending on the type
- the operator + has a different implementation determining which implementation should be used is based on the arguments only
- Coercion: allow the application of + to int and float. instead of defining + for all possible combinations of types, the arguments are automatically coerced this coercion may generate code (e.g. conversion from int to float) conversion is usually done towards more general types i.e. 5+0.5 has type float (since float ≥ int)

Type Inference

– ability to automatically deduce, either partially or fully, the type of an expression at compile time

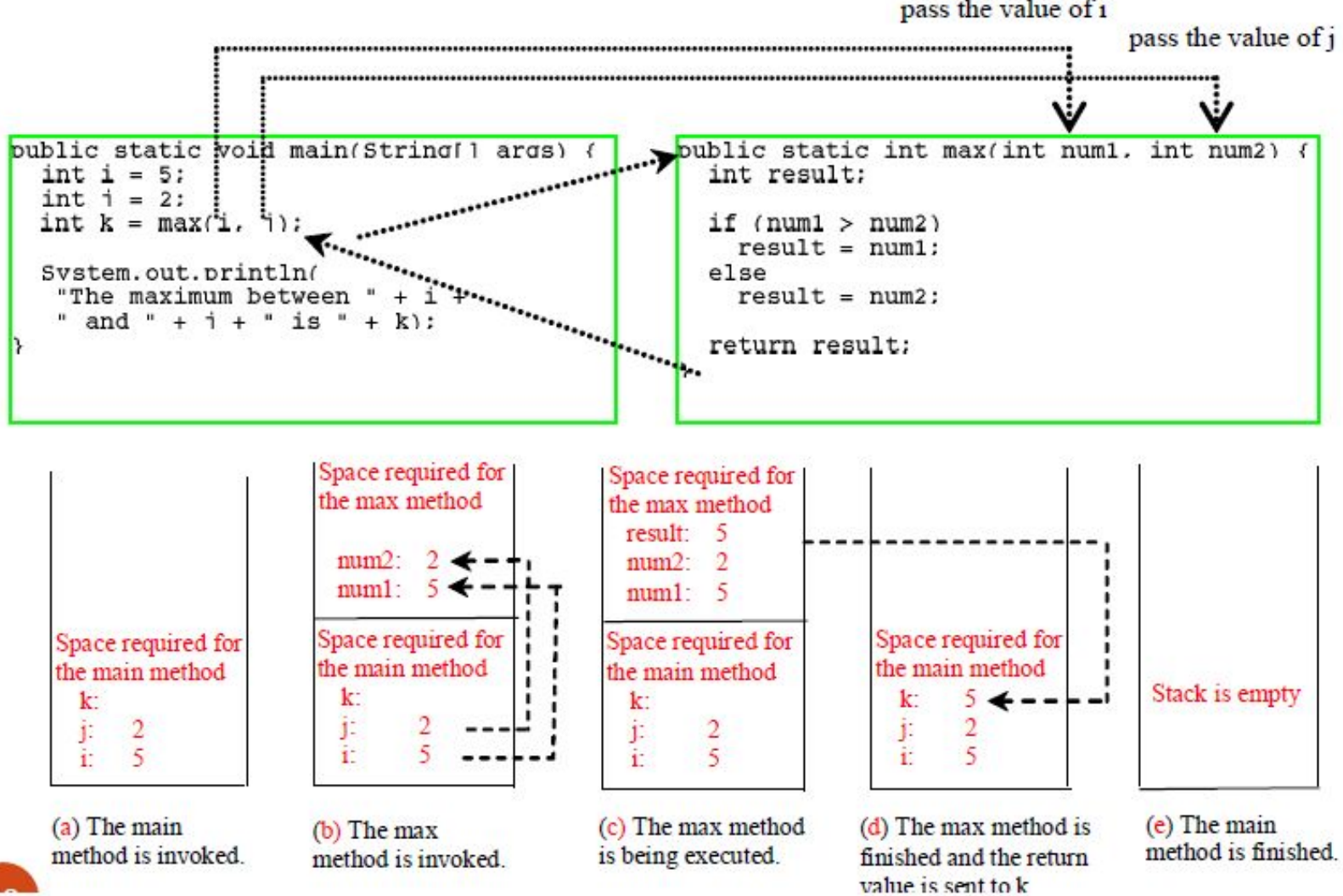
– the system will automatically infer that the header of ***f*** ought to have been fun f(x :: Boolean, y :: Number): ...

```
fun f(x, y):
  if x:
    y + 1
  else:
    y - 1
  end
End
```

Subroutine and Control Abstraction

- Subroutines are the principal mechanism for control abstraction in most programming languages.
- A subroutine performs its operation on behalf of a caller,who waits for the subroutine to finish before continuing execution.
- Most subroutines are parameterized: the caller passes arguments that influence the subroutine behavior, or provide it with data on which to operate.
- Arguments are also called actual parameters. They are mapped to the subroutine formal parameters at the time a call occurs.
- A subroutine that returns a value is usually called a function.
- A subroutine that does not return a value is usually called a procedure.
- Most languages require subroutines to be declared before they are used, though a few (including Fortran, C, and Lisp) do not. Declarations allow the compiler to verify

Stack Layout,
Calling sequence, parameter passing



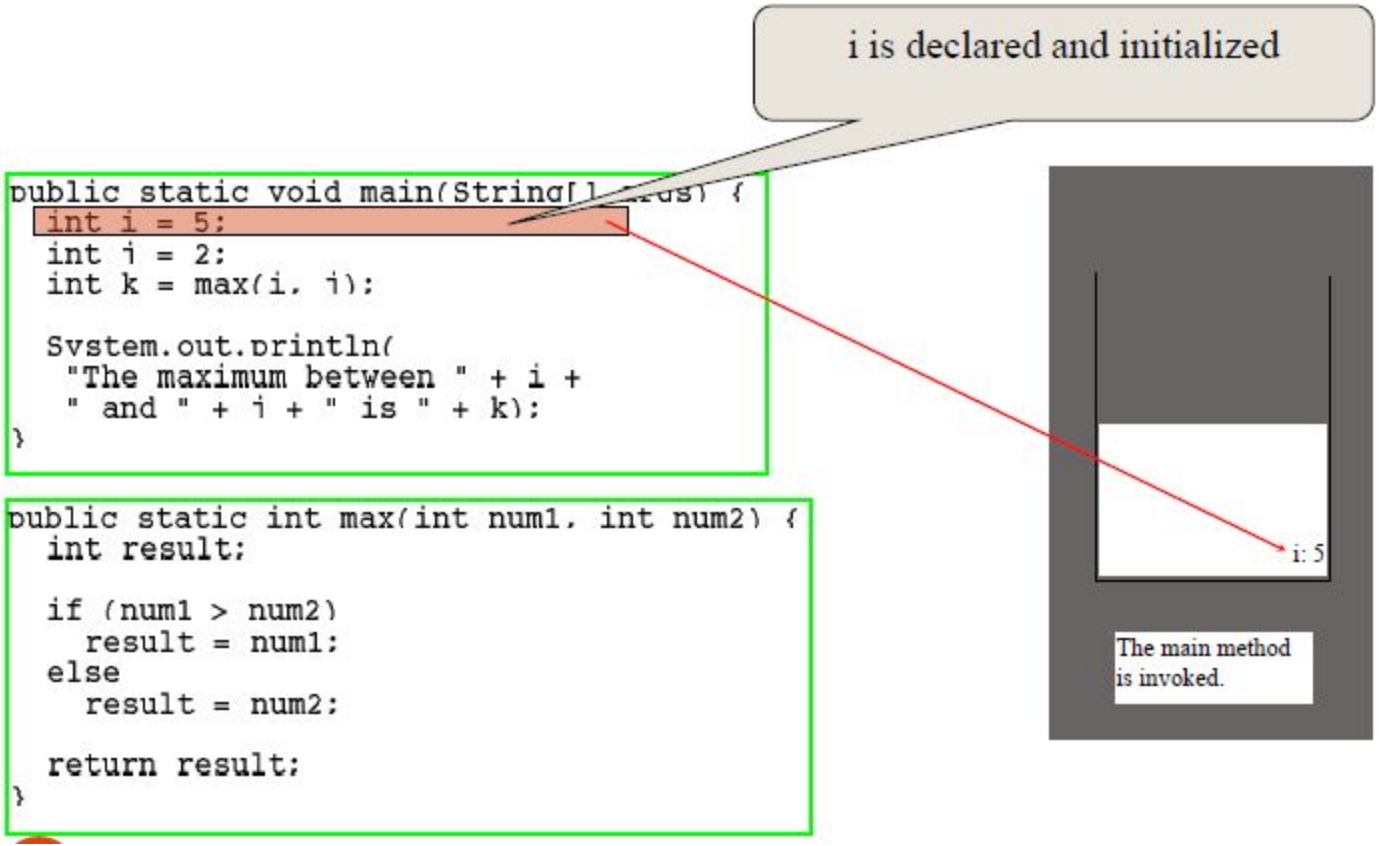
(a) The main method is invoked.

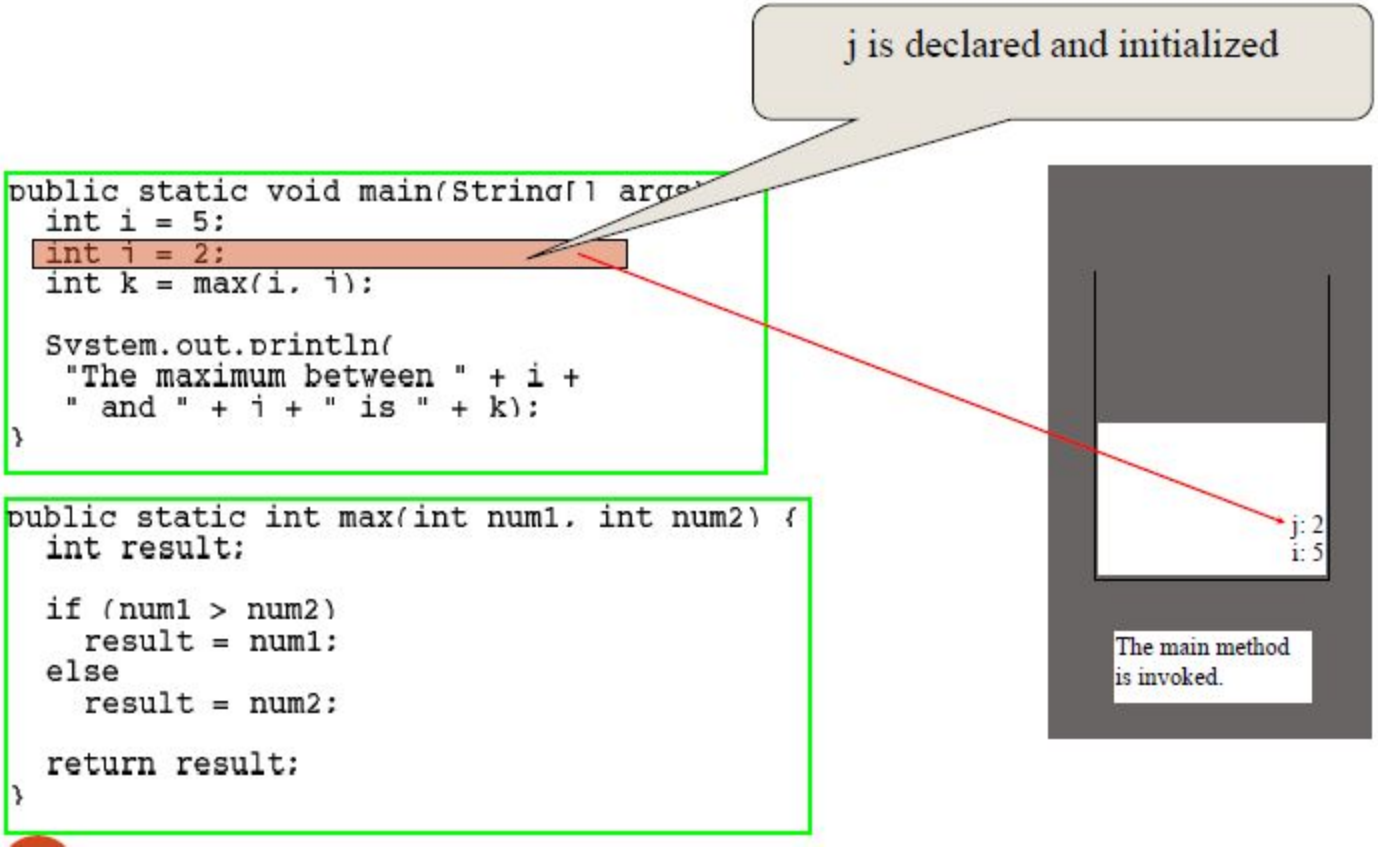
(b) The max method is invoked.

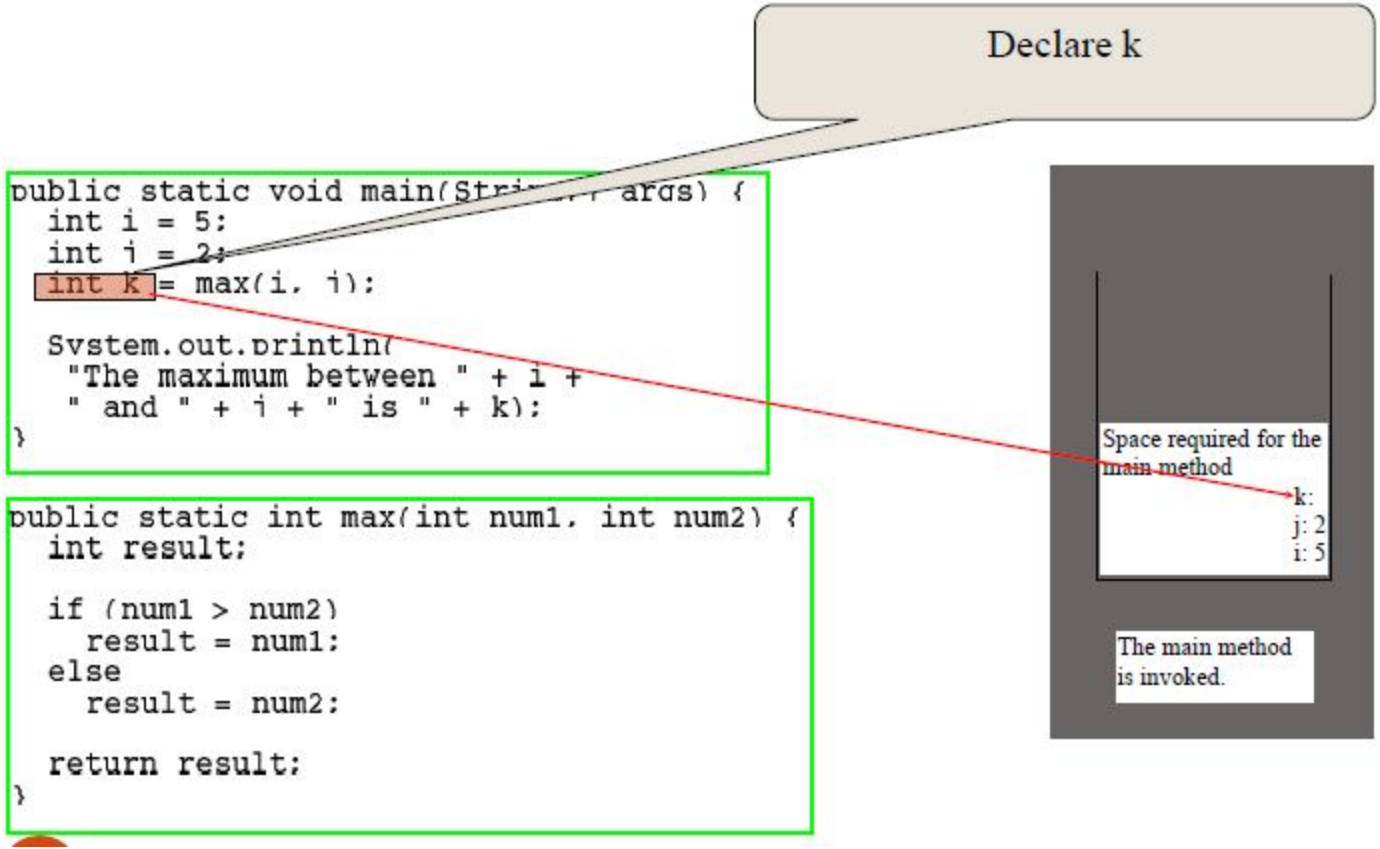
(c) The max method is being executed.

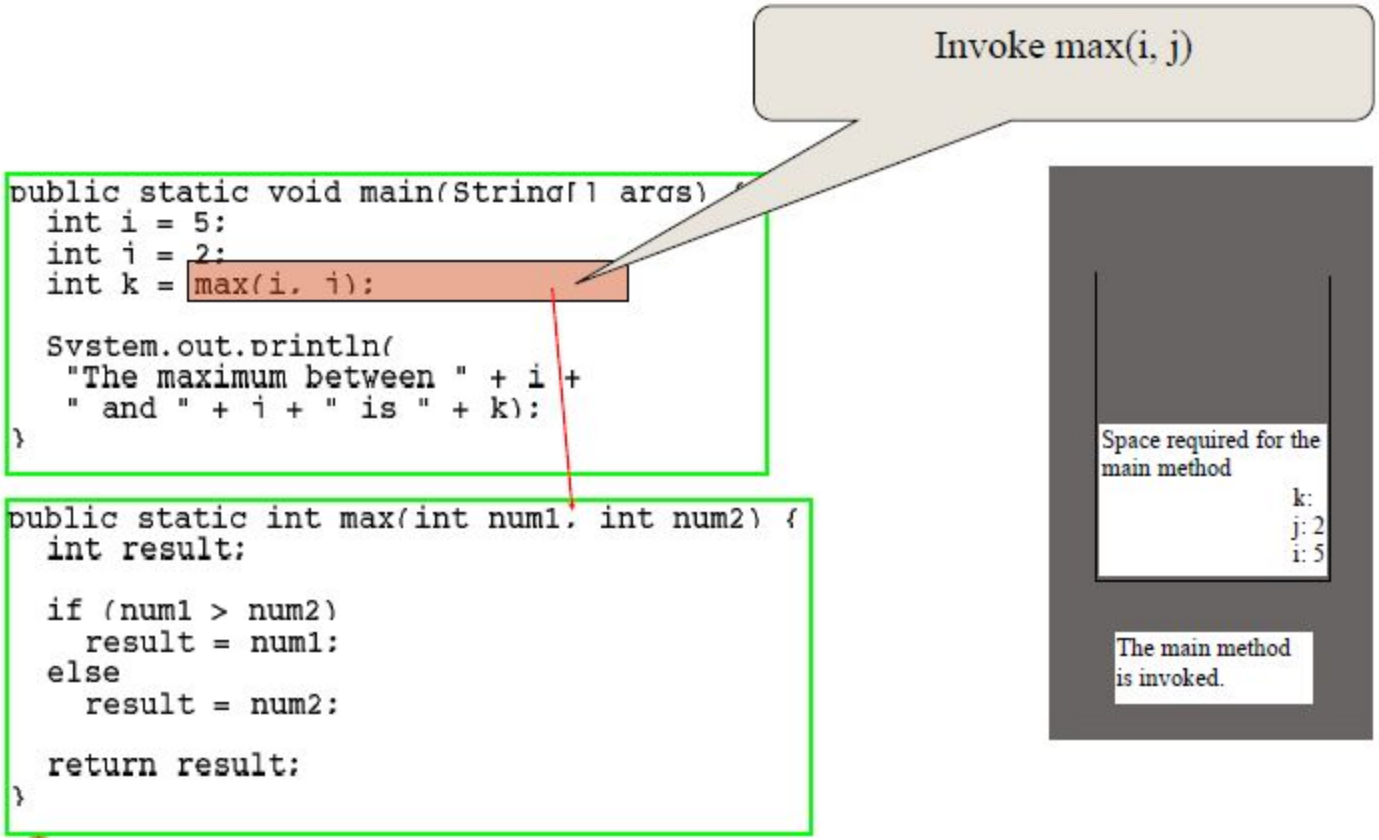
(d) The max method is finished and the return value is sent to k

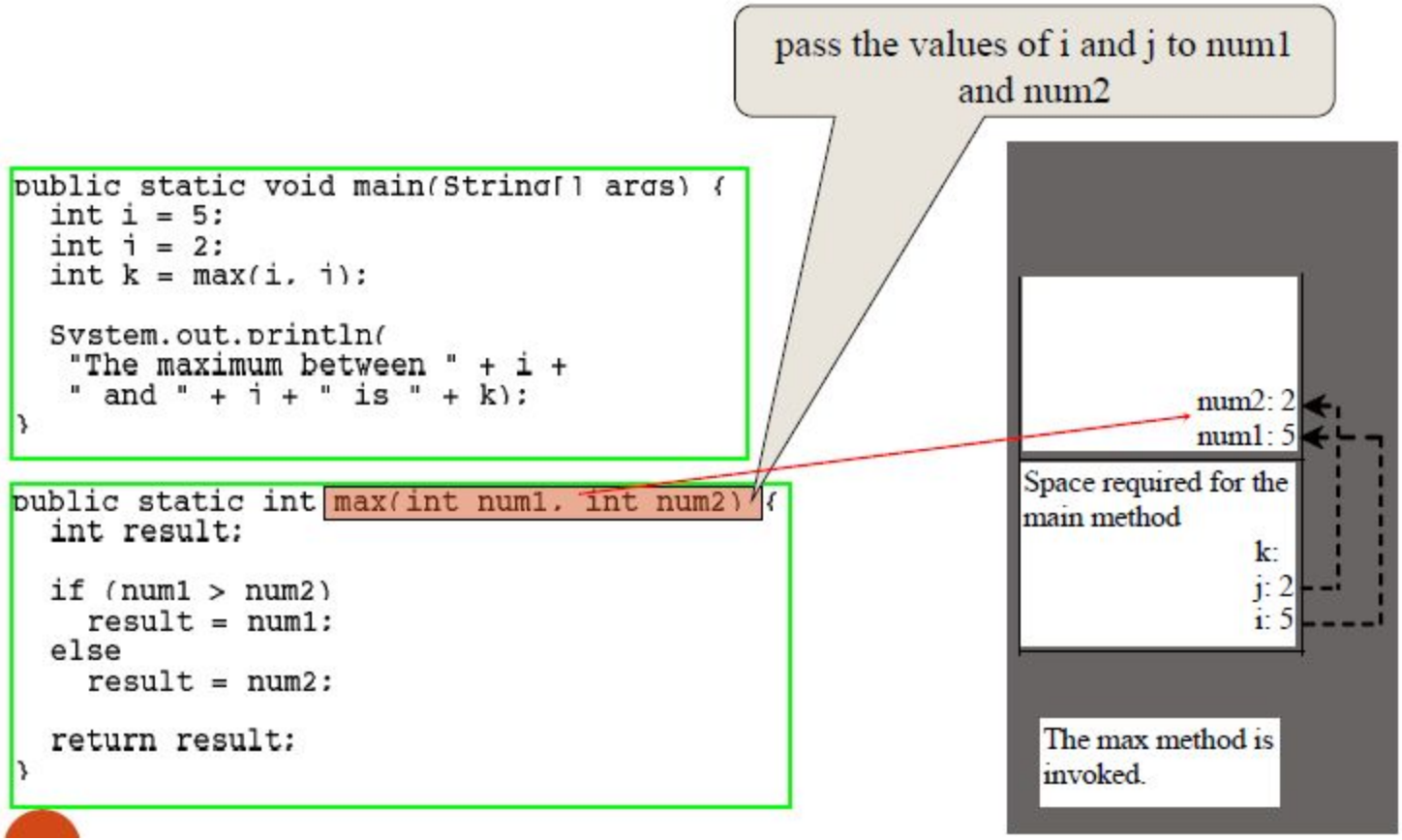
(e) The main method is finished.


i is declared and initialized
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
i: 5
The main method is invoked.


j is declared and initialized
public static void main(String[] args)
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
j: 2
i: 5
The main method
is invoked.

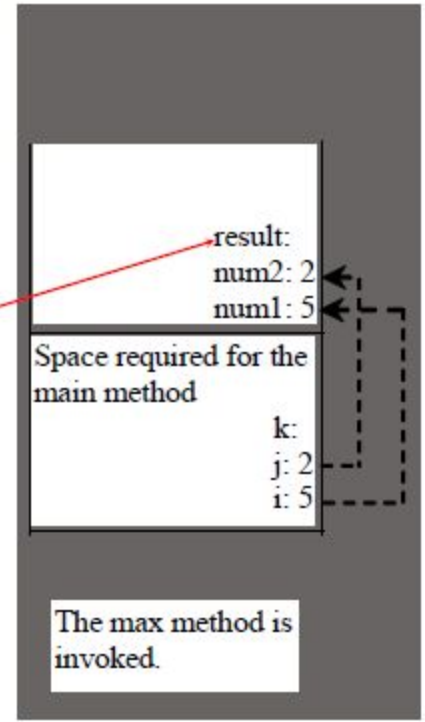

Declare k
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
Space required for the main method
k:
j: 2
i: 5
The main method is invoked.


Invoke max(i, j)
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
Space required for the main method
k:
j: 2
i: 5
The main method is invoked.


pass the values of i and j to num1 and num2
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
num2: 2
num1: 5
Space required for the main method
k:
j: 2
i: 5
The max method is invoked.

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

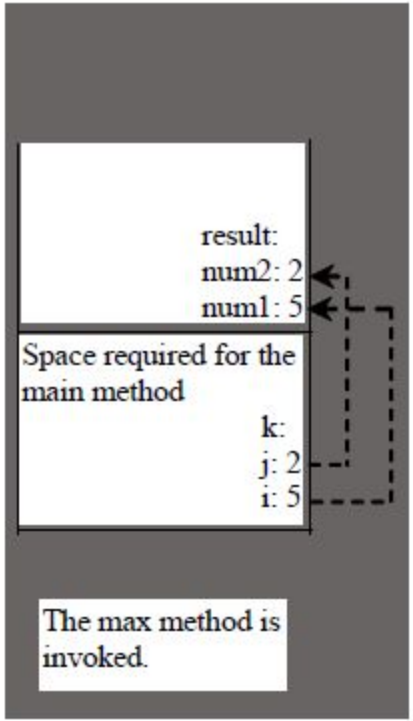
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

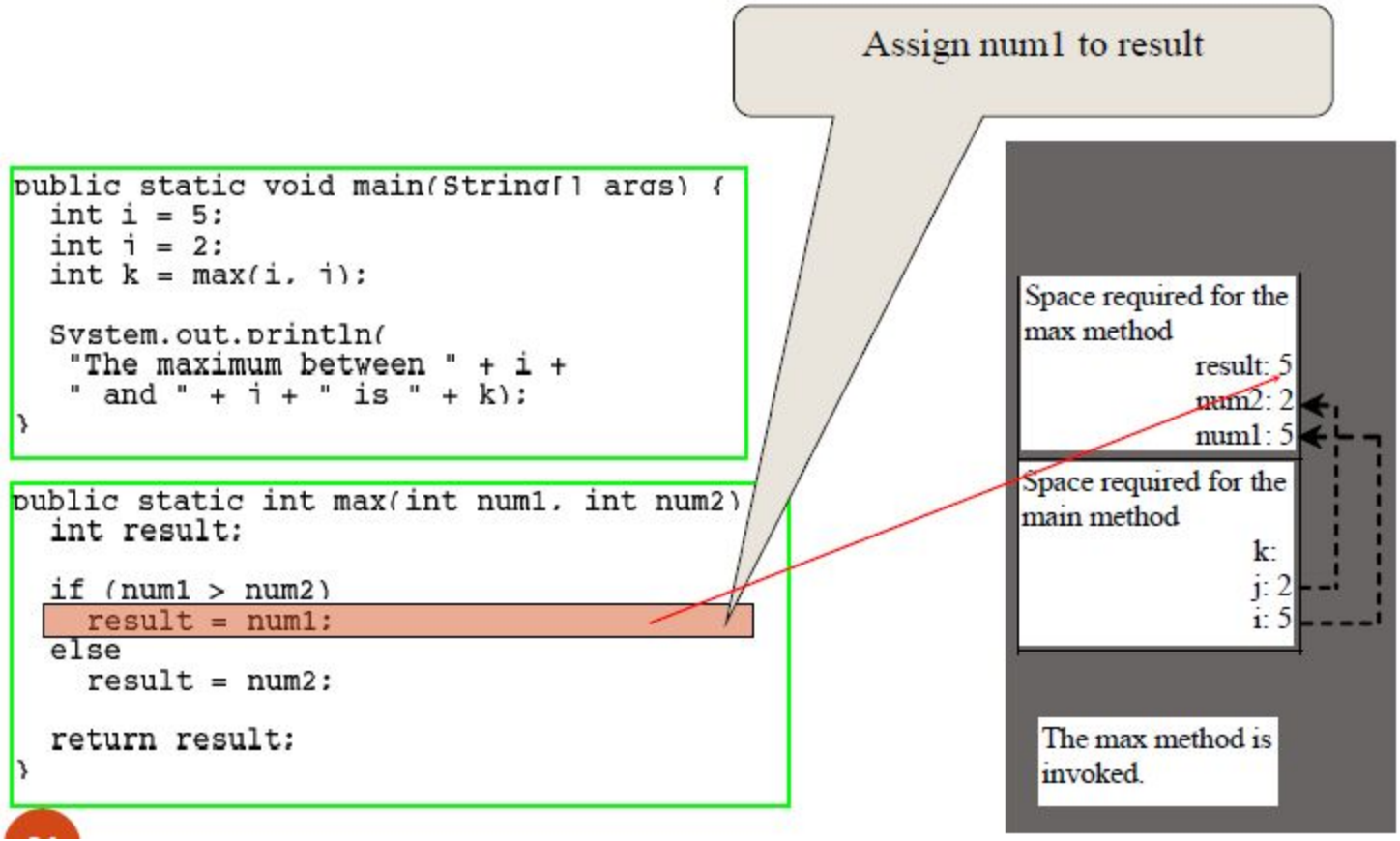


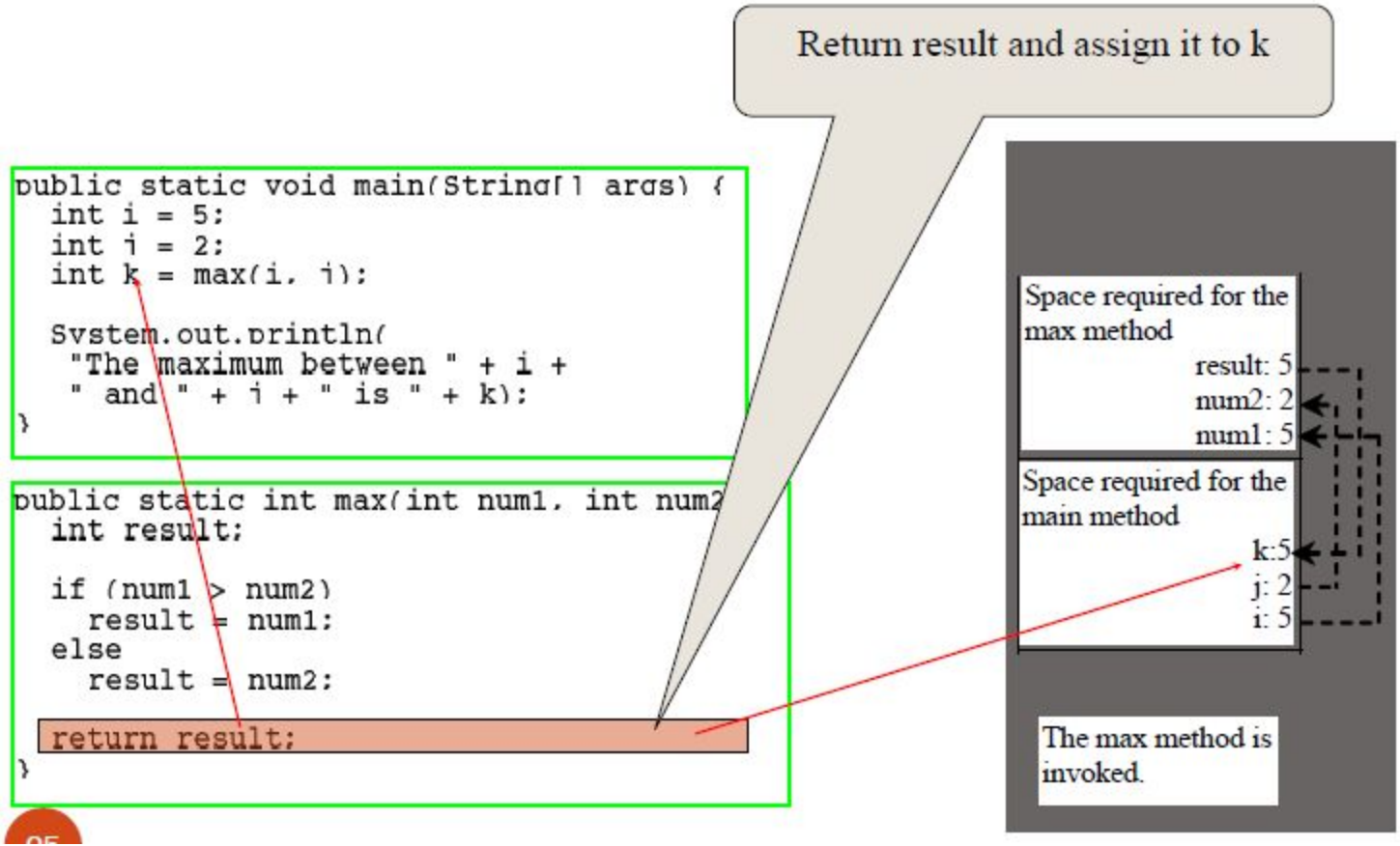
```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
    "The maximum between " + i +
    " and " + j + " is " + k);
}
```

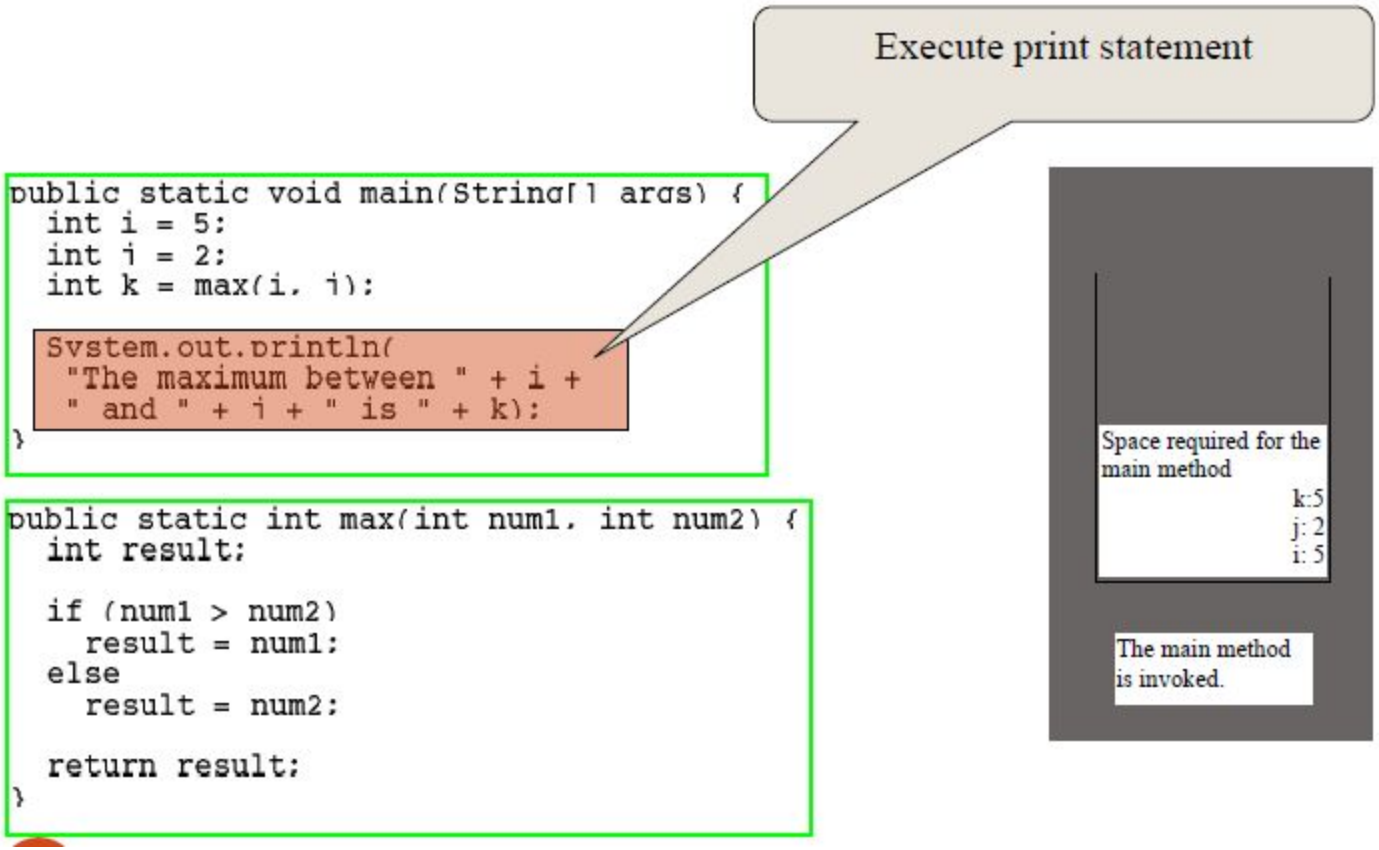
```
public static int max(int num1, int num2) {
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

(num1 > num2) is true




Assign num1 to result
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2)
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
Space required for the max method
result: 5
num2: 2
num1: 5
Space required for the main method
k:
j: 2
i: 5
The max method is invoked.


Return result and assign it to k
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
Space required for the max method
result: 5
num2: 2
num1: 5
Space required for the main method
k:5
j: 2
i: 5
The max method is invoked.


Execute print statement
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println(
"The maximum between " + i +
" and " + j + " is " + k);
}
public static int max(int num1, int num2) {
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
Space required for the main method
k:5
j: 2
i: 5
The main method is invoked.

- Reminder: maintenance of stack is responsibility of calling sequence and subroutines prologue and epilogue
- Optimizations:
  - space is saved by putting as much in the prolog and epilogue as possible
  - time may be saved by
    - putting stuff in the caller instead or
    - combining what's known in both places (interprocedural optimization)
    - Unfolding subroutines (e.g., Prolog/ Datalog unfolding optimizations)
- One cannot return references to objects on the stack
  - Rookie mistake in C: the lifetime is limited to function scope.

Stack pointers:

- The frame pointer (fp) register points to a known location within the frame of the current subroutine
  - fp usually points to the parameters (above the return address) for the current call
- The stack pointer sp) register points to the first unused location on the stack (or the last used location on some machines)
  - sp would point to where arguments would be for next call
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

Parameter Passing

- Most subroutines are parameterized: they take arguments that control certain aspects of their behavior, or specify the data on which they are to operate.
- Parameter names that appear in the declaration of a subroutine are knownas formal parameters.
- Variables and expressions that are passed to a subroutine in a particular call are known as actual parameters.
- Also called arguments

Prefix notation

- In scala

```
sum i1 i2 i3 ...
```

Infix notation

- ML allows the programmer to specify that certain names represent infix operators, which appear between a pair of arguments:

```
infixr 8 tothe;       (* exponentiation *)
fun x tothe 0 = 1.0
  | x tothe n = x * (x tothe(n-1));      (* assume n >= 0 *)
```

- Fortran 90 also allows the programmer to define new infix operators, but it requires their names to be bracketed with periods (e.g., A .cross. B), and it gives them all the same precedence. Smalltalk uses infix (or “mixfix”) notation (without precedence) for all its operations.

Parameter Modes

- Some languages—including C, Fortran, ML, and Lisp—define a single set of rules that apply to all parameters.
- Other languages, including Pascal, Modula, and Ada, provide two or more sets of rules, corresponding to different parameter-passing modes.
- As in many aspects of language design, the semantic details are heavily influenced by implementation issues.
- Different modes:
    - Call-by-value
    - Call-by-reference
    - Call-by-Sharing
    - Read-Only Parameters

- In Pascal, parameters are **passed by value** by default; they are passed by reference if preceded by the keyword var in their subroutine header's formal parameter list.
- Parameters in C are always **passed by value**, though the effect for arrays is passed by reference
- Neither call by value or reference option really makes sense in a language like Smalltalk, Lisp, ML, or Clu, in which a variable is already a reference, calls this mode **call-by-sharing**.
- Java uses call-by-value for variables of built-in type (all of which are values), and call-by-sharing for variables of user-defined class types (all of which are references).

Why call by reference?

- if the called routine is supposed to change the value of an actual parameter (argument), then the programmer must pass the parameter by reference.
- to ensure that the called routine cannot modify the argument, the programmer can pass the parameter by value.
- the implementation of value parameters requires copying actuals to formals, a potentially time-consuming operation when arguments are large. Reference parameters can be implemented simply by passing an address. (Of course, accessing a parameter that is passed by reference requires an extra level of indirection. If the parameter is used often enough, the cost of this indirection may outweigh the cost of copying the argument.)

Special-Purpose Parameters

- Default (Optional) Parameters
- Named Parameters

```
put(item => 37, base => 8);
```

- Variable Numbers of Arguments

```
int printf(char *format, ...)
{ ...
```

Generic Subroutines and Modules

- Subroutines provide a natural way to perform an operation for a variety of different object (parameter) values.
- In large programs, the need also often arises to perform an operation for a variety of different object types.
- An operating system, for example, tends to make heavy use of queues, to hold processes, memory descriptors, file buffers, device control blocks, and a host of other objects.
- The characteristics of the queue data structure are independent of the characteristics of the items placed in the queue.
- Generic modules or classes are particularly valuable for creating *containers*

- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right.
- A sorting routine, it needs to be able to tell when objects are smaller or larger than each other, but does not need to know anything else about them.
- Min routine
- Here **Implicit parametric polymorphism** could be applied but it makes the compiler substantially slower and more complicated. Alternative is **explicitly polymorphic**
- Generic modules or classes are particularly valuable for creating containers data abstractions that hold a collection of objects, but whose operations are generally oblivious to the type of those objects.
- Languages that support generic subrountines ar include Ada, C++ (which calls them templates), Clu, Eiffel, Modula-3, Java, and C#.

Exception Handling

- An exception can be defined as an unexpected—or at least unusual—condition that arises during program execution, and that cannot easily be handled in the local context.
- It may be detected automatically by the language implementation,or the program may raise it explicitly.
- Clu, Ada, Modula-3, Python, PHP, Ruby, C++, Java, C#, and ML,

```
try {
    ...
    if (something_unexpected)
        throw my_exception();
    ...
    cout << "everything's ok\n";
    ...
} catch (my_exception) {
    cout << "oops\n";
}
```

```
try {
    ...
    foo();
    ...
    cout << "everything's ok\n";
    ...
} catch (my_exception) {
    cout << "oops\n";
}
```

```
void foo() {
    ...
    if (something_unexpected)
        throw my_exception();
    ...
}
```

- Rule,
  - if an exception is not handled within the current subroutine, Propagation of an exception out of a called routine then the subroutine returns abruptly and the exception is raised at the point of call.
  - When an exception occurs in agiven block of code but cannot be handled locally, it is often important to declare a local handler that cleans up any resources allocated in the local block, and then "reraises" the exception, so that it will continue to propagate back to a handler that can (hopefully) recover.
  - if recovery is not possible, a handler can at least print a helpful error message before the program terminates.

Exception Propagation

```
try {                // try to read from file
    ...
    // potentially complicated sequence of operations
    // involving many calls to stream I/O routines
    ...
} catch(end_of_file) {
    ...
} catch(io_error e) {
    // handler for any io_error other than end_of_file
    ...
} catch(...) {
    // handler for any exception not previously named
    // (in this case, the triple-dot ellipsis is a valid C++ token;
    // it does not indicate missing code)
}
```

Coroutines

-

Events

- An event is something to which a running program (a process) needs to respond, but which occurs outside the program, at an unpredictable time.
- Common events are inputs to a
  - graphical user interface (GUI) system: keystrokes, mouse motions, button clicks.
  - network operations or other asynchronous I/O activity: the arrival of a message, the completion of a previously requested disk operation.
- Sequential Handlers
  - Need to change the traditionally, event handlers were implemented in sequential programming languages
  - synchronize access to such shared structures

- Thread-Based Handlers
  - events are often handled by a separate thread of control, rather than by spontaneous subroutine calls.
  - To protect the integrity of shared data structures, the main program and the handler thread(s) will generally require a full-fledged synchronization mechanism
  - C#, Java

Books

- Understanding Programming Languages, Ben Ari [Chapter 1]
- Scott M L, Programming Language Pragmatics, 3rd Edn., Morgan Kaufmann Publishers, 2009 [Chapters: 1(1.1-1.5), 3, 6, 7 (7.1.7.2,7.3) and 8]