Introduction

What is programming paradigm ?

- A programming paradigm is a fundamental style of computer programming.
- Paradigms differ in the concepts and methods used to represent the elements of a program (such as objects, functions, variables, constraints).
- And also steps that comprise a computation (such as assignations, evaluation, continuations, data flows).

Imperative programming paradigm

- Imperative programming is a programming paradigm that uses statements that change a program's state.
- In much the same way that the imperative mood in natural languages expresses commands.
- An imperative program consists of commands for the computer to perform.
- Imperative programs describe the details of *HOW* the results are to be obtained.
- HOW means describing the Inputs and describing how the Outputs are produced.
- Examples are: C, C++, Java, PHP, Python, Ruby etc.

Declarative programming paradigm

- Declarative programming is a programming paradigm—a style of building the structure and elements of computer programs—that expresses the logic of a computation without describing its control flow.
- Declarative programming focuses on *what* the program should accomplish.
- Declarative programming often considers programs as theories of a formal logic, and computations as deductions in that logic space.
- Examples are: SQL, XSQL (XMLSQL) etc.

Functional programming paradigm

- Functional programming is a subset of declarative programming.
- Programs written using this paradigm use functions, blocks of code intended to behave like mathematical functions.
- Functional languages discourage changes in the value of variables through assignment, making a great deal of use of recursion instead.
- Examples are : F#, Haskell, Lisp, Python, Ruby, JavaScript etc.

Object oriented programming paradigm

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods.
- There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type.
- In OOP, computer programs are designed by making them out of objects.
- Examples are: C++, C#, Java, PHP, Python.

Multi paradigm

- A multi-paradigm programming language is a programming language that supports more than one programming paradigm.
- The design goal of such languages is to allow programmers to use the most suitable programming style and associated language constructs for a given job.
- Languages such as C++, Java, Python are multi-paradigm programming languages that support object-oriented programming to a greater or lesser degree, typically in combination with imperative, procedural programming.

A Glance of different paradigms

Paradigm	Description	Examples
Imperative	Programs as statements that directly change computed state (datafields).	C, C++, Java, PHP, Python, Ruby.
Functional	Treats computation as the evaluation of mathematical functions avoiding state.	C++, Lisp, Python, JavaScript
Object-oriented	Treats datafields as objects manipulated through predefined methods only	C++, C#., Java, PHP, Python .
Declarative	Defines program logic, but not detailed control flow	SQL, CSS.

Conclusion

- There is still not a consensus (and probably there will never be) on a programming paradigm and a programming language is most suitable.
- All approaches have their advantages and disadvantages, with many supporting arguments and Case-studies.
- Despite that, it seems that nowadays the most popular paradigms for introductory courses are the procedural, with programming language C and procedural part of C++, the object-oriented, with languages Java and C++.

General comparison

Language	C	C++	Java	UNIX	Perl
Intended use	Application, system,general purpose, low-level operations	Application, system	Application, business, client-side, general, server-side, Web	File manipulation,print-ing text	Application, scripting, text processing, Web
Imperative	Yes	Yes	Yes	No	Yes
Object oriented	No	Yes	Yes	No	Yes
Functional	No	Yes	No	No	Yes
Procedural	Yes	Yes	No	No	Yes
Standardized ?	1989, ANSI C89,ISO C90,ISO C99,ISO C11	1998, ISO/IEC 1998, ISO/IEC 2003, ISO/IEC 2011	De facto standard via Java Language Specification	Filesystem Hierarchy Standard, POSIX standard	No

EVOLUTION

Language	Developer	Year	Derived from
UNIX	Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.	1969	Multics
C	Dennis Ritchie	1972	B
C++	Bjarne Stroustrup	1979	C
Perl	Larry Wall	1987	C
Java	James Gosling	1991	C++

LANGUAGE EVALUATION CRITERIA

READABILITY

Language	Characteristic SIMPLICITY	Illustration

C	Feature multiplicity	x++;
		++x;
		x = x+1;
		x += 1;

C++	Feature multiplicity Operator overloading	operator+ (addition & string concatenation)

Java	Feature multiplicity	x++;
		++x;
		x = x+1;
		x += 1;

Perl	Feature multiplicity	
UNIX	feature multiplicity No operator overloading	$x++; $(++x) x = $x+1 $x += 1

Language	Characteristic ORTHOGONALITY
C	Non orthogonal
C++	Non orthogonal
Java	Non orthogonal
Perl	Non orthogonal
UNIX	Orthogonal

Language	Characteristic CONCISE DATATYPES	Illustration
C	Truth value by some other data type No boolean data type	**int** flag =1;
C++	Boolean data type	**bool** flag =true;
Java	Boolean data type	**boolean** flag=false;
Perl	No boolean data type	$flag=1;
UNIX	No boolean data type	flag=1

Language	Characteristic SYNTAX DESIGN	Illustration
C	Compound statements Curly braces used. Form & meaning	Meaning of static depends on context
C++		Understandable if C is known
Java		Understandable if C/C++is known
Perl	Compound statements Curly braces used. Form & meaning	Prior knowledge is required Eg:split,shift,unshift etc
UNIX	No curly braces Prior knowledge is required	for var in 0 1 2 3 4 5 6 7 8 do echo $var done Eg:grep

WRITABILITY

Language	Characteristic SUPPORT FOR ABSTRACTION	Illustration

C	Supports abstraction	Returntype
		func_name(parameters)
		{
		}

C++		modifier returnType nameOfMethod (Parameter List) { // method body }
Java		

Perl	Supports abstraction	sub fun_name
		{
		}

UNIX	Supports abstraction	function name( )
		{
		}

Language	Characteristic EXPRESSIVITY	Illustration
C	Shorthand operators Shortcircuit operators Loops	+= && for loop
C++		
Java		
Perl		
UNIX	Shorthand operators Shortcircuit operators Loops	++ && for loop

RELIABILITY

Language	Characteristic TYPE CHECKING	Illustration
C	Strongly typed	int a=5;
C++	Strongly typed	int a=5;
Java	Strongly typed	int a=5;
Perl	Loosely typed Runtime	$a=5; $a=“hii”;
UNIX	Loosely typed Runtime	a=5;

Language	Characteristic EXCEPTION HANDLING	Illustration
C	No exception handling	

C++	Exception handling exists	try
		{
		// protected code
		}catch(ExceptionName e)
		{
		// code to handle
		ExceptionName exception
		}

Java	Exception handling exists	try { //Protected code }
		catch(ExceptionType1 e1) {
		//}
		finally { //The finally block
		always executes. }

Perl	Exception handling exists	chdir('/etc') or warn "Can't change directory"; chdir('/etc') or die "Can't change directory";
UNIX	No exception handling	

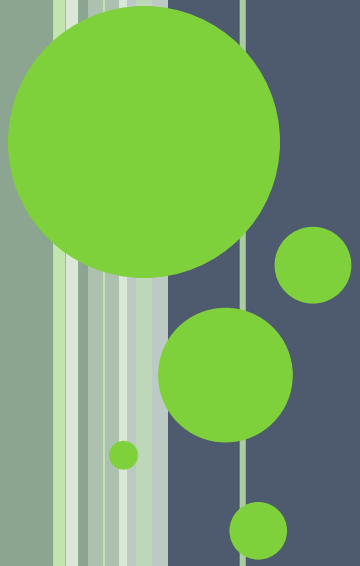
Language	Characteristic ALIASING	Illustration
C	Aliasing using pointers	int A[10]; int *p= A;
C++	Aliasing using reference variables	float total=100; float &sum=total;
Java	Aliasing using object references	Square box1 = new Square (0, 0, 100, 200); Square box2 = box1;
Perl	Aliasing using reference variables	$r=\@arr;
UNIX	Giving a name for a command	alias dir=ls

Binding

Language	Type binding	Storage binding
C	Static binding int a;	Local variables are stack dynamic by default. Static variables are declared using keyword static Explicit heap dynamic variables using malloc()
C++	Static binding int a;	Local variables are stack dynamic by default. Static variables are declared using keyword static Explicit heap dynamic variables using new
Java		

language	Type binding	Storage Binding
Perl	Dynamic binding $a=5;	implicit declarations and explicit declarations with static scoping, lots of heap bindings (for strings, arrays, etc.), run-time memory manager does garbage collection, implicit allocation and de-allocation
UNIX	Dynamic binding a=5	Local variables are stack dynamic by default.

DATA TYPES



Language	Primitive	Derived	User defined
C	int, float, char, double, void	pointer,function, array,structure, union	structure, union, enum, typedef
C++	void,char,int,float, double,bool	Array, function,pointer, reference	structure, union, enum, typedef, class
Java	int, float, long, double, char ,boolean,short,byte	Array, function,pointer, reference	structure, union, enum, typedef, class,sequence
Perl	Scalar, hash, array	function	No user defined datatypes
UNIX	No primitive datatypes	Array,function	No user defined datatypes

VARIABLES & CONSTANTS

VARIABLE & CONSTANT DECLARATION

LANGUAGE	VARIABLE	CONSTANT	TYPE SYNONYM
C	type name «**=** initial value»**;**	**enum{** name **=** value **};**	**typedef** type synonym**;**
C++	type name «**=** initial value»**;**	**const** type name **=** value**;**	**typedef** type synonym**;**
Java	type name «**=** initial_value»**;**	**final** type name **=** value**;**	NA
UNIX	name**=**initial_value	NAME ="Zara Ali" readonly NAME	NA
Perl	«**my**» **$**name «**=** initial_value**»;**	**use constant** name **=>** value**;**	NA

SCOPE

LANGUAGE	SCOPE
C	Static scoping
C++	Static scoping
Java	Static scoping
UNIX	Static scoping
Perl	Static scoping

Assignment statements

LANGUAGE	ASSIGNMENT

C	c = a;
	a = a + b;
	a+ = b;

C++	
Java	
UNIX	a=$b
Perl	$a=$b;

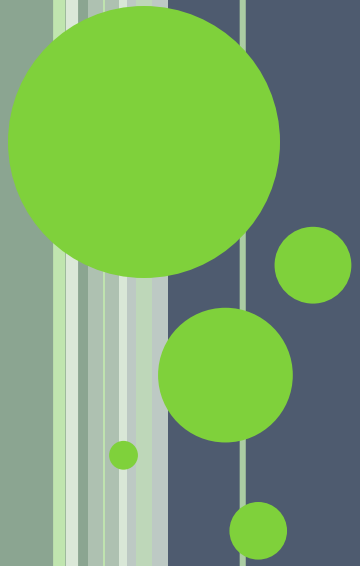
Arithmetic expressions

LANGUAGE	ARITHMETIC EXPRESSIONS

C	c = a + b;
	c = a - b;
	c = a * b;
	c = a / b;
	c = a % b;
	c = a++;
	c = a--;

C++	
Java	
UNIX	val=`expr $a + $b` val=`expr $a - $b` val=`expr $a \* $b`
Perl	$c = $a + $b; $c = $a - $b; $c = $a * $b; $c = $a / $b; $c = $a % $b; $c = $a ** $b;

Control flow



CONDITIONAL STATEMENTS

Language	if	else if	case	conditional operator
C C++ Java	**if (**condition**)** **{**instructions**}** «**else** **{**instructions**}**»	**if (**condition**)** **{**instructions**}** **else if** **(**condition**)** **{**instructions**}** ... «**else** **{**instructions**}**»	**switch** **(**variable**)** **{case** case1**:** instructions «**break;**» ... «**default:** instructions»**}**	condition**?** valueIfTrue **:** valueIfFalse
UNIX	**if** condition- **then** expression «**else** expression» **fi**	**if** condition- **then** expression **elif** condition- **then** expression «**else** expression» **fi**	**case** "$variable " **in** "$condition1") command;; "$condition2") command;; **esac**	

Language	If	Else if	Case	Conditional operator
Perl	**if (**condition**)** **{**instructions**}** «**else** **{**instructions**}**» or **unless (**not condition**)** **{**instructions**}** «**else** **{**instructions**}**»	**if (**condition**)** **{**instructions**}** **elsif (**condition**)** **{**instructions**}** ... «**else** **{**instructions**}**» or **unless** **(**notcondition**)** **{**instructions**}** **elsif (**condition**)** **{**instructions**}** ... «**else** **{**instructions**}**»	**use feature "switch";** ... **given (**variable**)** **{when (**case1**)** **{** instructions **}** ... «**default** **{** instructions **}»}**	condition**?** valueIfTrue **:** valueIf- False

LOOP STATEMENTS

Language	while	Do while	For	foreach
C	**while (**condition**) {**instructions **}**	**do {** instructions **} while (**condition**)**	**for (**«type» i **=** first**;** i **<=** last**; ++**i**) {** instructions **}**	NA
C++				«**std::**» **for_each(**start**,**end**,**function**)**
Java				**for (**type item **:** set**) {**instructions **}**
UNIX	**while** [condition] **do** Instructions **done** or **until** [notcondition] **do** Instructions **done**	NA	**for ((**i **=** first**;** i **<=** last**; ++**i**)) do** Instructions **done**	**for** item **in** set **do** Instructions **done**

Language	While	Do while	For	foreach
Perl	**while (**condition**)** **{**instructions **}** or **until** **(**notcondition**) {**instructions **}**	**do {** instructions **}** **while (**condition**)** or **do {** instructions **}** **until (**notcondition**)**	**for**«**each**» «**$i**»**(**first **..**last**)** **{**instructions **}** or **for ($i =** first**; $i <=** last**; $i++)** **{**instructions **}**	**for**«**each**» «$item» **(**set**)** **{**instructions **}**

Other control flow statements

Language	Exit block(break)	Continue	Label	Branch(goto)
C	**break;**	**continue;**	label**:**	**goto** label**;**
C++	**break;**	**continue;**	label**:**	**goto** label**;**
Java	**break** «label»**;**	**continue** «label»**;**	label**:**	NA
UNIX	**break** «levels»**;**	**continue** «levels»**;**	NA	NA
Perl	**last** «label»**;**	**next** «label»**;**	label**:**	**goto** label**;**