Serdica J. Computing **5** (2011), 129–152

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

TWO-LANGUAGE, TWO-PARADIGM INTRODUCTORY COMPUTING CURRICULUM MODEL AND ITS IMPLEMENTATION

Vladimir Zanev, Atanas Radenski

Abstract. This paper analyzes difficulties with the introduction of object-oriented concepts in introductory computing education and then proposes a two-language, two-paradigm curriculum model that alleviates such difficulties. Our two-language, two-paradigm curriculum model begins with teaching imperative programming using Python programming language, continues with teaching object-oriented computing using Java, and concludes with teaching object-oriented data structures with Java.

1. Introduction. In the mid-1990s, object orientation–analysis, design, and programming–firmly established itself as a mainstream methodology in the software development industry. The industry found convincing advantages of object orientation in many areas, such as:

- User requirements analysis and modeling
- Software design

ACM Computing Classification System (1998): K.3.2.
Key words: Computer science education, teaching programming, CS1, CS2, Java, Python.

- Developing of well-structured and reliable applications
- Software maintenance and extensibility, code reuse, application robustness; resilience of code changes
- Abstraction, encapsulation, and information hiding

Nowadays, programming languages that provide direct linguistic support for object orientation, such as Java, C++, C#, and Visual Basic are dominant in the industry and the demand for knowledgeable and skillful programmers with degrees in computer science or computer engineering is very high. The TIOBE Programming Community Index [18] provides a monthly ranking of the most widely used programming languages in computer education and industry. According to the February 2011 TIOBE Index, the first eight most popular programming languages are: Java (17.8%), C (15.8%), C++ (8.8%), PHP (7.8%), Python (6.3%), C# (6.3%), Visual Basic (5.9%) and Objective-C (3.0%).

Universities in the USA, Europe, and worldwide have quickly followed the software industry's shift to object orientation and have redesigned their computing curricula to provide students with an early exposure to OOP [19]. In the process, OOP replaced the older imperative programming paradigm in the computing curricula. Soon, OOP became the most popular paradigm in introductory undergraduate computer science courses: Computer Science 1, 2, and possibly 3, commonly abbreviated as CS1, CS2, and CS3. In this process, Java has become the dominant language in introductory computing curricula [25]. Hence, we focus on Java in our discussions of OOP-related educational issues.

While the need to teach OOP is widely accepted, how exactly to do it has been a subject of intensive debates through the years. Advocates of the objects-first approach in particular insist on putting the stress on object-orientation exclusively and from the very beginning [20], while opponents express concerns with some harmful effects of the objects-first approach in introductory computing courses [21]. We are among those who are concerned with the drawbacks of the objects-first approach.

The OOP paradigm is based on a number of fundamental concepts, such as objects and classes; interfaces; encapsulation; information hiding; inheritance. Most of these OOP concepts, if not all, may seem too abstract and complicated to beginners in computing. Teaching OOP from the very beginning (in accordance with the objects-first idea) in CS1 courses shifts attention away from some fundamental programming principles and techniques and obscures the algorithmic and problem solving nature of computer science. Students are forced to immediately

confront the complexities of OOP and master the sophisticated Java language syntax and semantics, even for the simplest of programs. The objects-first approach can easily lead students to believe that computer science courses are about learning how to program in Java and less about the foundational ideas and techniques of the discipline. The complexities of teaching OOP often mean that core computing ideas, like iteration, flow of control, and procedures/functions are given less emphasis. Our experience is that students in upper level computer science courses who have been subjected to the objects-first approach often lack basic problem-solving skills that should have been acquired in CS1.

Difficulties with the objects-first approach and Java in CS1 motivated us to develop and experiment with an alternative approach to the introductory computing course sequence, an approach that defers Java and OOP from CS1 to CS2/CS3, and uses Python and imperative programming in CS1. We develop this theme in the rest of the paper. Section 2 offers a critical analysis, from educational perspective, of the *de facto* standard first Java program that appears in the majority of CS1 Java-based textbooks. Section 3 continues this critical analysis with an outline of a range of pedagogical issues with Java. Section 4 discusses possible approaches to the introductory computing curriculum, with the focus on recent ACM recommendations. Section 5 presents our model curriculum, founded on "imperative-first with Python, objects-second with Java" approach. Section 6 describes an implementation of this model curriculum and presents evidence for its success. Section 7 offers concluding remarks.

2. First Java program – critical analysis. While Java is certainly not the only OOP language, it is entrenched in both academia and industry; it is also a fairly typical representative of compiled OOP languages which continue to dominate the OOP realm [18].

With the introduction of the very first Java topic, the first line of Java code, and the first Java program, most instructors recognize they have a serious pedagogical burden (and many students also quickly understand that they are headed to serious difficulties). This is why: at the initial stages of an introductory Java-based course, instructors cannot fully explain even a simple program completely because of its inherent complexity and unavoidable advanced features. Relatively complex technical OOP concepts that cannot be avoided even in the simplest initial Java programs are tightly connected, interrelated, and cannot be taught or learned independently, in a reasonable linear sequence. The unavoid-

able initial complexity of Java can easily decrease the student motivation and make students believe that OOP cannot be quickly mastered.

Consider, for example, the code of a typical "Hello World" Java program (Fig. 1).

```
1  public class Hello
2  {
3      public static void main(String[ ] args)
4      {
5          // display Hello greeting in the console window
6
7          System.out.println("Hello World");
8      }
9  }
```

Fig. 1. Source code of "Hello World" Java program. Numbers on the left side of the source code lines are not part of the code

The first couple of lines of Java code involve a bewildering number of technical concepts that call for clarification by the instructor when the program is explained to a beginner student: access modifiers, class, visibility modifiers, methods, method return types, parameters, data types, scope, and arrays. If a logical, systematic method of instruction is to be followed, such clarification would be expected even before the discussion of the body of the main method.

In the rest of this section, we examine, line by line, how the "Hello World" program elements are explained in three widely adopted Java textbooks that have appeared in multiple editions: *Big Java* by Cay Horstmann [22]; *Java. How To Program* by H. Deitel and P. Deitel [23], and *Java Software Solutions* by J. Lewis and W. Loftus [24].To facilitate our analysis, we decorate the "Hello World" Java program by adding labels – which are simply circled numbers – to the source code (Fig. 2). In the following critical analysis, we use numeric labels from Fig. 2 to refer to specific Java entities.

Line 1: public class Hello. This line includes the following Java language elements:

- ① public**:** an access modifier
- ② class**:** a keyword used to specify the beginning of a class definition.

In order to explain access modification to a beginner, the textbook and the instructor will have to explain the idea of access, the types of access supported in

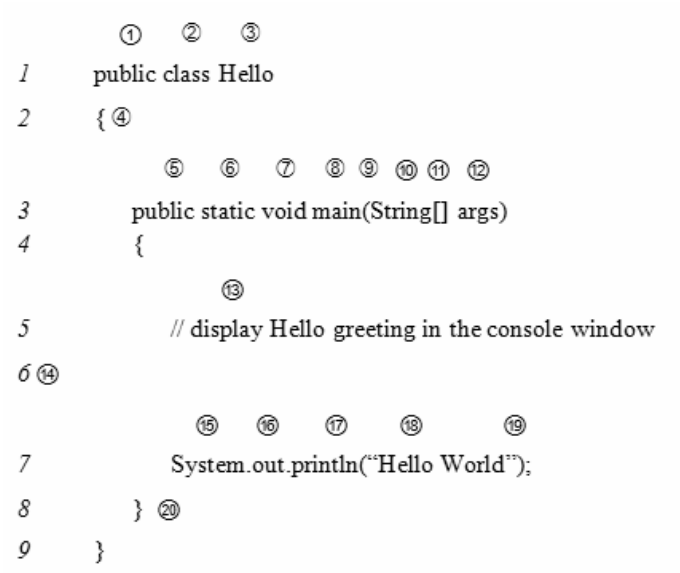

Fig. 2. Decorated version of the "Hello World" Java program

Java, and where access modifiers can be used. One can state that access modifiers can be specified with Java classes and methods, but at this initial stage of the course of study, students do not know what a Java class is, and what a method is.

An explanation of the ="class" keyword requires the clarification of a variety of concepts, usually covered in more than one textbook chapter and spanning several class sessions. For example, answers to the following questions, to mention a few, need to be provided in an explanation attempt: What is a Java class? How do we design and develop classes? Why do we need classes? Any answers to such questions are likely to seem incomprehensible for the beginner.

The *Big Java* textbook [22] offers the following explanation of what a class is: "At this point, you should simply regard the line **public class *ClassName* {}** as a necessary part of the "plumbing" that is required to write any Java program". Other textbooks [23, 24] simply skip any explanations about what a class is, omit any discussion about access modifiers, and simply give brief informal characterizations: "The rest of the program is class definition"; "the line begins a class definition".

Line 3: public static void main(String[ ] args). This line involves the following Java concepts:

- ⑤ access modifiers (see discussion above)
- ⑥ static modifiers
- ⑦ return types
- ⑧ main method
- ⑩ data types and String data type
- ⑪ arrays
- ⑫ parameters and arguments.

In order to explain static modifiers, one has to explain what a class object is and how the static modifier affects methods, constructors, and variables. A convincing explanation of method parameters, arguments, and return values demands a fairly complete explanation of class methods as a whole. The introduction of the special main method is yet another challenge, and any attempted explanation may trigger more questions that give answers, such as why the main method has a special name. Data types, strings, and arrays are complex concepts that are covered in separate textbook chapters and discussed in a number of class sessions. The *Java Software Solutions* textbook [24] offers the following superficial explanation of the Java elements in line 3: "The *main* method definition in a Java program is always preceded by the words *public, static,* and *void*, which we examine later in the text. The use of *String* and *args* does not come into play in this particular program. We describe these later also". The *Java How to Program* textbook [23] uses a similar superficial approach: "... main is a program building block called method. ... Methods are explained later in Chapter 6. For now, simply mimic main's first line in your Java applications." The *Big Java* textbook [22] explains the main method in a similar way: "At this time, simply consider *public class CalssName* {*public static void main(String[] args)* {...}} as yet another part of "plumbing"."

Line 7: System.out.println("Hello World"). This line involves the following Java elements:

- ⑮ System**:** a reference to the System class
- ⑯ out**:** an object from the System class
- ⑰ println**:** a method of the object out; parameters of the println method; how to print out different types of data

An explanation of what the "System" class is may involve other concepts, such as default import of the java.lang package. An explanation of the "out" object triggers the need to define the class instance concept; in addition, such an ex-

planation may need to clarify how instances of static classes are used and what the difference is between static and non-static classes. An explanation of the "println" method involves dot-notation for object instances and "pritnln" parameters. Textbooks [22, 23, and 24] seem to offer adequate introductions of the above concepts. However, our experience is that beginners have difficulties grasping the notion of package and comprehending the purpose of a special object "out" that is a part of a special "System" class; unfortunately, textbooks do not help much in this respect. The *Java Software Solutions* textbook [24], for example, offers this deferred explanation: "The *System.out* object represents an output device or file, which by default is the monitor screen. To be more precise, the object's name is out and it is stored in the *System* class. We explore that relationship in more details at appropriate point in the text."

Lines 2, 4, 5, 8, and 9. Even relatively simple syntax elements, such as delimiters (in Java—curly braces: ④ and ⑳, or parentheses ⑨, names ③, comments ⑬, empty lines ⑭, and case-sensitivity are difficult to explain without reference to some more complicated language elements. Most notably, the explanation of the purpose of curly braces, for example, needs to involve blocks and scoping.

In brief, using a typical OOP language such as Java in the CS1 course requires the immediate introduction of numerous underlying concepts such as classes, objects, methods, parameters, arguments, access modifiers, data types, and arrays. These concepts are in fact very complex, interrelated, and most likely impenetrable for a beginner. As a result, students may quickly lose interest and motivation in Computer Science as a discipline.

3. Pedagogical issues with the Java language and platform.

In recognition of the numerous difficulties of teaching CS1 and CS2 with Java as an introductory programming language, the ACM Education Board created in 2004 the ACM Java Task Force (JTF). The JTF was to review and identify the problems of teaching Java, and to develop pedagogical resources (Java libraries) to more easily teach CS1 and CS2. In 2006, the JTF published the JTF Final Report [4] with a list of characteristics Java and its API that causing significant pedagogical problems. Some – but far from all – of these issues have alleviated with the design of Java 5 in September 2004. Unfortunately, a significant list of Java shortcomings – from pedagogical perspective – cannot be eliminated simply because they are grounded in the Java language design strategy in general, and

the language syntax and semantics in particular. In this section, we enumerate and briefly discuss only those issues identified by the JTF [4] that still remain concerns from the perspective of introductory computing curricula.

1. Scale and Instability
   - Scale

     One very serious problem facing instructors and student is the enormous size and complexity of Java libraries; size and complexity make it difficult for beginners to use libraries in writing useful programs. The problem with the scale is aggravated with each new release of Java which, as a rule, modifies and extends existing libraries, and also introduces new ones.
   - Instability

     Java developers continue to respond to the needs and demands of the computer industry by frequent changes of the Java platform, thus imposing continuing changes of teaching resources.
2. Linguistics Complexity
   - Complexity of Static methods and classes

     The early exposure of beginners to the ubiquitous "main" method and to static classes such as "Math" and "System" is confusing for students. Curiously, at this stage some students tend to create programs in which all methods are static; this practice undermines the object-oriented paradigm itself.
   - Complexity of Exceptions

     In Java, exceptions are objects; exception handling is based on dynamic binding of propagating exception to handlers. Exception propagation and handling rules are complicated and code that involves exception handling can easily be regarded by beginners as difficult and even unreadable. For example, it is difficult to explain to beginners the purpose and usefulness of exception handlers that catch exceptions without any executable code (i.e., exception handlers that merely "swallow" the exception).
3. APIs Issues
   - Lack of simple input mechanism in early Java releases

     This issue was partly alleviated with the introduction of the

java.util.Scanner class in Java 5.

- Conceptual difficulty of the Java graphics model

  The following three properties of the Java graphics model create considerable pedagogical difficulties: forgetful bitmaps (each Java graphics component has the responsibility to respond to update events); stateless graphics contexts (Java graphics contexts do not maintain state from call to call); unfamiliar definition of the graphics coordinate system, where the origin is located in the upper-left corner space, with x values increasing to the right and y values increasing downward.

- Difficult-to-use GUI API

  The standard Java GUI classes provided by the Swing library are not easy to use for novices.

- Inadequate support for event-driven and concurrent programming

  Java's event and concurrency models are too complex for entry-level curricula.

4. Core Syntax and Semantics

   The core Java language syntax and semantics are not easy to teach and learn. The following pedagogical difficulties are very often encountered in Java-based CS1 classes [9].

   - Semicolons

     Java textbooks and instructors alike often make statements like this: "In Java every statement has to end with a semicolon". The problem is that students must not literally apply this rule to a number of Java statements, such as if, for, and while statements**.** When a student puts a semicolon at the end of for/while statements, his code will compile but its execution will result in an infinite loop (Fig. 3). When a student puts a semicolon at the end of a one-way selection if statement (not followed by an else clause), her code will compile but will still be logically incorrect. If she puts a semicolon at the end of an else clause, the result will be a possibly confusing compilation error, 'else' without 'if'.

```
1  int sum = 0;
2  for (int i=0; i < 10; i++);
3  {
4      sum = sum + i;
5  }
6  System.out.println(sum); // prints out 10 instead of 45
```

Fig. 3. Problematic use of a semicolon at the end of a loop statement

- Type Casting

  Type casting in Java is a complicated topic to teach to beginners. It requires good understanding of primitive and reference types, and also familiarity with inheritance of classes and interfaces. A variety of type casting, such as up-casting and down-casting, need to be discussed. There are static (compile-time) casting rules and dynamic (runtime) casting rules. While Java 5 reduced the complexity of casting with the introduction of boxing and un-boxing, it added new extreme complexities to casting with the introduction of generics.

- Arithmetic

  Java supports arithmetic by means of an ample set of integer operations over a variety of data types: byte, short, int, long, float, and double. Most arithmetic operators, such as +, -, and * are straightforward and pose no challenges to beginners. In contrast to these, the integer division (/) and the remainder (%) operators are a source for student programming errors and confusion. A typical student error is to use division of int values without proper casting. This is illustrated by the following example student code: int cents = 995; float dollars = 19.99; float totalInDollars = cents/100 + dollars; /* totalInDollars becomes 28.99 instead of 29.94*/.

Java 5 and Java 6 releases enhanced the Java platform with valuable additions, such as wrapper classes, generics, enumerations, and iterators. These releases have aimed at faster execution; easier integration with JavaScript; simpler GUI–design they even permitted strings in switch statements. Some of these valuable additions are still underrepresented in the Java textbook market.

All Java language changes and enhancements have been subjected to a full backward compatibility requirement. This requirement led to some ad hoc solutions that were patched over the original Java language and became subject of strong criticism. The long awaited introduction of generics with Java 5 in

particular is practical but not devoid of serious flaws, such as the conversion of Java from a type-safe to a type unsafe language for example [14, 16].

4. Planning, designing, and developing introductory computer science courses. Teaching computer science, especially introductory computer science courses, means teaching both science and technology. Planning, designing, and developing of introductory computer science courses (CS1/CS2/ CS3) requires careful consideration of many factors, requirements, and constraints including: curriculum standards; software industry expectations; demands of prospective employers; degree type; program goals; student body and student characteristics; accreditation and assessment challenges; traditions and culture of computing departments; textbooks and IDEs availability; and others. The process of planning, designing, and development of introductory computer science courses requires effort and is based on experience and knowledge of multiple faculty members at different levels: undergraduate curriculum committee, department, college, university, and even state level.

The world's leading computing professional organization – the Association for Computing Machinery (ACM) and the IEEE Computer Society–have been systematically engaged in the development of computing curricula [1, 2, 3], focusing on principles, course structure, and curriculum implementation strategies. Recent curricula releases in the USA include the ACM Computing Curricula 2001 [1] (CC2001CS) and its interim revision, Computer Science Curriculum 2008 [3] (CSC2008). In Europe, the Bologna Process [6] defined high level standards and protocols for European university degree programs in general. Regretfully, the Bologna Process has not yet provided recommendations that are specifically targeted at higher education in computing.

CC2001CS [1] describes and suggests six possible approaches to introductory computer science courses:

- Imperative-first
- Objects-first
- Functional-first
- Breadth-first
- Algorithms-first
- Hardware-first

In addition to these six approaches, a couple of other known models of

introductory programming courses should be mentioned – logical-first (characterized by programming with relations and inference) and concepts-first. The concepts-first approach provides fundamentals for programming in multiple programming paradigms: procedural, logical, functional; object-oriented; parallel and distributed programming, script programming; and event-driven programming. Nowadays, the OOP paradigm is ubiquitous in the CS1, CS1, and CS3 courses worldwide, and it is used to implement different pedagogical approaches, such as objects-first, objects-late, or even imperative-first.

It is interesting that CC2001CS considers and recommends different programming paradigms and approaches (not necessarily OOP) for CS1, and that CS2 and even CS3 courses are only implicitly based on the OOP paradigm.

Choosing a successful introductory programming paradigm is, naturally, an important decision that directly affects students, instructors, and all computing-related academic programs, such as computer science (CS), computer engineering (CE), software engineering (SE), computer information systems (CIS), and information technology (IT). In the introductory computer science curriculum in particular, the choice of the first programming paradigm – and the choice of the first programming language as well – have immediate impact on student motivation, student retention rate, and – in the long run – on the development of competitive computing professionals – programmers, Web developers, database developers, software engineers, system analysts, and others.

5. A model curriculum: imperative-first with Python, object oriented-second with Java approach. The choice of a proper programming paradigm and the related choice of a first programming language are of significant importance for CS, CE, CIS, and IT programs. These choices have profound impact on students' ability to acquire fundamental computing concepts; on the development of computational thinking and problem-solving skills; and on the preparedness of students for upper-level studies in computing.

We believe that in CS1, teaching Java as a first programming language is problematic in objects-first and objects-late approaches alike; this belief is grounded on our personal teaching experiences and also on literature research. We have therefore investigated alternative CC2001CS proposals for alternative pedagogical approaches – and programming languages other than Java – in introductory CS courses [1]. Most notably, CC2001CS proposes the following approaches as alternatives to objects-first: imperative-first, functions-first, breadth-

first, algorithms-first, and hardware-first.

The breadth-first approach provides a framework for an introductory CS course that includes introduction and topics from different computing disciplines – computers and programming, programming languages, databases, operating systems, Web development, and others. The purpose of an introductory CS1 course based on the breath-first approach is to give the student a broad view of computer science and to prepare him/her for a well-informed decision on whether to pursue further studies in computing.

We perceive algorithms-first and hardware-first approaches as more conceptual and less practical. In contrast, the functions-first approach, although solidly grounded in theory, goes beyond conceptual knowledge to support practical problem-solving. Recall that the functional programming paradigm originated in the mathematical theory of lambda-calculus. Functional programming languages support function definitions and permit function applications, with recursion being the principal programming technique. Students develop algorithmic thinking and problem-solving skills by means of functions and recursion and avoid the complexity and abstractions of the object-oriented paradigm. Indeed, functional programming languages are very orthogonal, with few basic notions, and easy to learn.

Despite the well-recognized strengths of the functions-first approach to introductory computing courses, this approach generates some concerns. Older functional programming languages, such as Lisp in general and Scheme in particular, are well known for poor readability due to the large number of nested parentheses needed. Other concerns stem from the vast number of built-in functions that are difficult to comprehend and use properly in Lisp-like programs. On the positive side, newer functional languages as Standard ML, Haskell, and Scala departed from Lisp in their syntax, and are also richer semantically. Such functional languages, most notably Haskell, have been actually used as introductory languages in the CS curricula in some universities, and have been promoted as introductory languages because of their capability to build mathematical maturity and abstract thinking ability [27]. Yet, functional languages are still not widespread in industry and can easily be viewed as impractical by students and administrators in education alike. Last but not least, pure functional languages do not support changes in state in a straightforward manner and only simulate state with pure functional mechanisms (such as Haskell's monads). However, such simulations can be complex and difficult to understand, particularly by beginners, in contrast to the clear and direct support of state changes provided by

non-functional languages.

The functions-first approach has been usually materialized in introductory computing courses by means of the Scheme language. The Schemers server [17] maintains a list of over 100 universities and colleges worldwide that have adopted a Scheme-based functions-first approach in introductory computing courses. A comparison of recent lists to previous years' lists hints that the members of this list are decreasing. For example, previously well-known functional programming with Scheme CS1 courses taught at MIT and Georgia Tech have been dropped from the list because these universities changed their introductory computing curricula.

The concepts-first approach provides a basis for programming in a variety of programming paradigms: procedural, object-oriented, logical, functional, and concurrent. This approach is implemented by means of a set of specialized kernel languages. A kernel language consists of a small number of programming significant concepts and is used in the study of one specific programming paradigm. Developed and published by the Mozart Consortium [10], this approach never reached popularity in computing programs because of the lack of good pedagogical resources and tools, such as IDEs and textbooks, and because of the lack of support from the software industry as well.

The imperative programming paradigm, being the oldest programming paradigm, is very well understood from both practical and theoretical perspective, and is widely accepted in software development. It directly reflects the dominant von Neumann computer architecture and is based on the notions of memory (state) and commands. Imperative programming involves a few simple constructs: sequence of statements; selection and repetition; functions and procedures[1]. Yet, the availability of procedures permits relatively complex application structures to be expressed by hierarchical decomposition into simpler procedural structures. Hierarchical decomposition provides a valuable enhancement to the style and coding techniques of the students.

We are among those who prefer to use the imperative-first approach in introductory computing education. With an imperative-first approach, students can learn from the very beginning how to think algorithmically and how to solve problems programmatically. For an imperative-first approach, the first course – even if it is taught using an object-oriented language – focuses on the im-

[1]Some authors exclude procedures and functions from imperative programming and use the name procedural programming for an extension of the imperative paradigm with procedures. We are among those who consider "imperative" and "procedural" to be synonyms.

perative aspects of that language: expressions, control structures, procedures, and functions, and other core elements of the traditional imperative paradigm. The techniques of object-oriented programming are covered in the follow-on CS2 course [1].

Note that using an imperative-first approach in a CS1 course does not necessarily exclude the study of objects whatsoever. It only means putting the stress on – and devoting enough time for fundamental imperative programming structures, such as flow of control, selection, iteration, functions and procedures. Later in the course, it can be only natural – and even necessary – to introduce principal objects that are predefined in the programming language [11]. This is easy to do and actually unavoidable with some languages, such as Python and Ruby.

The choice of the first programming paradigm used to teach introductory computing courses cannot be separated from the choice of the first programming language since we have to deliver the content of the programming paradigm in an appropriate form. After careful evaluation of different programming languages (Java, Python, PHP, Haskell, C++, Scheme) in 2003-2004, we settled on Python. We continue to believe that Python is the best programming language, for the time being, to use as first programming language. We are not the only ones to have reached this conclusion: an ever increasing number of universities in the last few years have adopted Python in their introductory computer science courses [8, 15].

We prefer Python as a first language for a number of reasons, including the following ones:

1. Python is a general-purpose high-level programming language that is applicable to real-life problems; yet Python was initially designed as a programming language for education [12].
2. Python is supported in computer industry (extensively used in Google, Yahoo, Nokia, and the US Government for different type of software projects).
3. Python is a multi-paradigm programming language. It supports in full OOP, imperative, structured programming, and even functional programming to some extent [5].
4. Python is beginner-friendly and easy to learn. It is a highly readable language with a visual layout using whitespace indentation instead of curly braces or keywords in other programming languages.
5. Python is an interpretive and interactive programming language which al-

lows students to run segments of code and understand how the program code is working.

Python has several implementations, such as CPython, JPython, and IronPython. CPython is the reference Python implementation, free and open source, and unquestionably the most popular one. A significant number of IDE support Python, including free open source implementations and also commercial implementations. The Integrated Development Learning Environment (IDLE) is packaged within the standard Python distributions for Windows, Mac OS X, and Linux/Unix. The popular NetBeans and Eclipse IDEs also support Python with plug-ins. A number of Python textbooks, both introductory and advanced, are also available, including free online textbooks and documentation.

Our preliminary evaluation of Python in 2003-2004 [11], our subsequent multi-year teaching experience, and the positive student feedback that we have received convince us that an imperative-first approach with Python is successful and beneficial for student learning in introductory CS1 courses.

In education, when it comes to follow-up courses, however, the object-oriented paradigm should not be ignored [7]. Nowadays, object-orientation dominates the practice of software analysis requirements, design, testing, and development [18]. Object-orientation practices lead to reliable and robust software, reusable components, extensibility, and good maintainability. The object-oriented paradigm is beneficial in the development of a variety of software products: desktop applications; Web applications and services; mobile applications; games; embedded applications; and databases for example. There is a significant demand in the software industry for professionals with object-oriented knowledge and skills. Our preferred choice of programming language to teach CS2 - and also CS3 in cases when the introductory sequence consists of three, rather than two courses - is Java. Java is a mainstream object-oriented language which has been so popular in the last decade as to maintain the number one spot of the Tiobe index [18]. The availability of a wide selection of pedagogical resources and tools, such as textbooks and free entry-level IDEs (such as BlueJ, DrJava, and JGrasp, to mention a few), makes Java a proper choice for CS2. We recognize that beyond Java, other popular multi- paradigm languages such as Python, C#, and C++ could also be used as a teaching vehicle in CS2. In CS2 – and in CS3 when applicable - we prefer Java to Python because Java exposes students to a typical compiled OOP language, after their exposure to an interpreted language, Python, in CS1; this provides students with knowledge of the two principal programming

language cultures. In addition, we prefer Java to C# because C# is limited to Microsoft environments only, while Java is usable in virtually any platform. Finally, we prefer Java to C++ because C++ is not as type-safe as Java is.

Table 1. Introductory Computing Sequence

Course	**Main Subject**	**Programming Paradigm**	**Programming Language**
CS1	Programming Fundamentals	Imperative	Python
CS2	Object-Oriented Programming	Object-oriented	Java
CS3	Data Structure and Algorithms	Object-oriented	Java

6. Implementation and evaluation of the model curricula. The authors of this paper have implemented a two-language, two-paradigm introductory computing curricula in California's Chapman University and in Georgia's Columbus State University. Our CS1 courses focus on imperative programming with Python, CS2 introduce object-oriented programming with Java, and CS3 courses teach basic data structures with Java. This approach has been productively used since the mid-2000's [11].

Our CS1 courses employ Python to cover the following topics:

1. Introduction to Computing
2. Computing Basics
   - 2.1. Control Structures: Selection
   - 2.2. Control Structures: Repetition
   - 2.3. Functions
3. Object-Based Computing
   - 3.1. Data Collections: Lists and Dictionaries
   - 3.2. Strings, Files, and the Web
   - 3.3. Graphics and Interfaces
4. Introduction to OO Programming
   - 4.1. Objects and Classes
   - 4.2. Inheritance

Our CS2 courses employ Java to cover the following topics:

1. Introduction to Object-Oriented Computing

2. Java Language Structures
3. Primary Objects: Characters; Strings; Numbers; Arrays; Vectors
4. Classes and Inheritance: Constructors; Instance Variables; Methods; Super and Subclasses; Nested Classes
5. Interfaces and Packages
6. Exceptions
7. Concurrency: Threads and Synchronization
8. Input-Output: Files; Streams; Serialization
9. User Interfaces: Swing; Components and Containers: Layout Management; Events
10. Graphics: 2D Graphics; Rendering; Shapes, Texts, and Fonts

Our CS3 courses use Java to introduce core data structures, such as lists, trees, hashes, and graphs. At the same time, CS3 courses provide students with an opportunity to further develop and strengthen their Java knowledge and OOP skills.

Our introductory computing courses are supported with comprehensive online study packs [11, 12, and 13]. The study packs support each course topic by means of complete self-contained resources (such as e-texts, sample programs, tutorial links, and slides) and activities (such as programming assignments and quizzes). Student outcomes are tested with conceptual and problem-solving exams. For practical work, students use entry-level IDEs: IDLE for Python in CS1; DrJava and BlueJ for Java in CS2. In CS3, students can elect to use advanced IDEs as alternatives to the simpler CS2 IDEs.

At Chapman University, the CS1/CS2/CS3 sequence forms the core of the undergraduate computer science program. In addition, CS1/CS2 courses are required for students majoring in mathematics. The CS1 course is required for the biology major and is also a general education quantitative reasoning option. While these courses are offered onsite, they have been supported by online study packs hosted at `http://studypack.com`.

At Columbus State University, the content of the CS1/CS2/CS3 sequence is packed in a remedial post-graduate course. This is a bridge-type course that is required for acceptance in the computer science master's program for students without computing background. This-one semester course covers the contents of a CS1/CS1/CS3 course sequence by allocating five weeks for CS1-level introduction to computing with Python, seven weeks for CS2-level object-oriented computing

with Java, and three weeks for CS3 data structures and algorithms with Java. Students are expected to complete the course with a grade of at least B. The course has been offered online with the support of CS1/CS2/CS3 study packs at `http://studypack.com`.

Various end-of-semester student surveys provide evidence for the success of our two-language introductory curriculum approach. The rest of this section outlines results from some of the surveys; additional surveys that are not discussed in this paper are in line with those presented below. All survey responses are on a 5-point scale from 1 (least important) to 5 (most important).

Table 2 contains data from a CS1 student survey at Chapman University [11]. As seen in Table 2, students strongly support the choice of Python as a CS1 language.

Table 2. Python evaluation

Select answers to these questions	
Would you recommend Python as a first language to beginners in computing?	4.5
Would you recommend Python programming to others?	4.1
Would you recommend another language, not Python, as a first language to beginners?	2.4

Table 3 presents results from a student survey at Columbus State University. This survey has been offered to students who have completed the entire CS1/CS2/CS3 sequence, as incorporated in a semester-long postgraduate online course. As seen from Table 3, students at Columbus State University strongly prefer Python as a first language. They also recognize the importance of studying Java. Table 3 shows that Python helped students to learn Java.

Verbal comments from various surveys have helped us refine the implementation of our courses. They have also provided additional testimony for the benefits of our two-language, two-paradigm introductory computing curriculum model. Anonymous Columbus State University students for example have provided this input:

- "I have enjoyed the second part of the class [*CS2 with Java*] very much. I'll have to admit that I like Python more than Java but I realize the importance of a structured language like Java. I would like to keep practicing with Java and go back and do the optional labs when I have time so that I can keep learning and not forget what I have learned so far. I have enjoyed this very much."

Table 3. Evaluation of the CPSC 6106 course

No.	Question	Average
Part 1. Python		
1.	Would you prefer Python as a first programming language?	4.7
2.	Would you recommend Python programming language to other students?	4.2
3.	Python programming language evaluation criteria (readability, coding, productivity)	4.8
Part 2. Java		
1.	Your preliminary study of Python has helped you learn Java	4.7
2.	Do you have to study Java programming?	3.9
3.	Would you recommend Java as a first programming language?	1.5
4.	Java programming language evaluation criteria (readability, coding, productivity)	3.7
Part 3. Data Structures		
1.	I understand the implementation and functionality of linear and non-linear data strictures	3.8
2.	I can use and develop programs to search in and sort data structures	4.0
The course as a whole		
1.	I have progress in my ability to solve problems, think critically, and make decisions	4.7
2.	The course was academically challenging	4.3
3.	I can articulate the core concepts or content of this course	4.4
4.	The content of this course is valuable	4.7
5.	Instructor's entire performance evaluation	4.2
6.	The teaching materials (textbook, slides) are useful	4.6
7.	The assignments/labs are useful	4.5
8.	The quizzes and exams are appropriate and useful	4.4

- "The class has been fun. I feel like I have learned what the CS department wanted to me to learn with this class. I will tell you that I rose to the challenge to stay up with the pace of the course."
- "Python was a nice intro into programming. I would have liked to see some more real world application of it and better understand how it is used on the web, but I do not think that was part of the objective for the class."

7. Conclusion. This paper presents our research, design, implementation, and teaching experience in the area of introductory computer science courses

– CS1, CS2, and CS3. We have developed a two-language, two-paradigm curriculum model that consists of three components: CS1, Imperative programming with Python; CS2, Object-oriented computing with Java; and CS3, Object-oriented data structures and algorithms with Java.

We have implemented and successfully used this curriculum model at our home institutions since the mid 2000's. Our curriculum model is based on a three-course sequence, rather than two-course sequence, because a three-course sequence is exactly what is recommended by the ACM Curriculum Guidelines for Undergraduate Degree Programs in Computer Science [1]. The same two-language, two-paradigm approach, however, can be incorporated in a two-course sequence in programs with very high degree of rigor.

We have investigated the current introductory computing curricula in six leading US schools: Massachusetts Institute of Technology, Stanford University, California Institute of Technology, Cornell University, Princeton University, and Georgia Institute of Technology. Three of these six universities - Massachusetts Institute of Technology, Georgia Institute of Technology, and California Institute of Technology are now offering two-language introductory course sequences: CS1, Imperative programming with Python; and CS2, Object-oriented computing with Java. In August 2010, Carnegie Mellon University (CMU) published a report for major revisions to its introductory Computer Science education [26]. According to the report, the CMUs plan is to use Python for imperative programming in their introductory CS 15–110 course, and Java for OOP in Software System Construction 15–214 course. In this respect, the CMUs plan resembles our two-language, two-paradigm curriculum model.

In conclusion, more and more universities are switching to two-language, two-paradigm curriculum model in their introductory Computer Science courses. We perceive this recent trend as a testimony to the soundness of our proposed introductory Computer Science curriculum model. In particular, universities adopt imperative programming first with Python and object-oriented second with Java approach in introductory Computer science education with the expectation of improving student motivation, satisfaction, retention, and passing rate.

The authors would like to thank the anonymous reviewers for their insightful comments and recommendations.

REFERENCES

[1] ACM Computing Curriculum 2001. Curriculum Guidelines for Undergraduate Degree Programs in Computer Science. `http://www.acm.org/education/education/education/curric_vols/cc2001.pdf, 2011`

[2] ACM Computing Curriculum 2005. The Overview Report. `http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf, 2011`

[3] ACM Computer Science Curriculum 2008. An Interim Revision of CS 2001. `http://www.acm.org//education/curricula/ComputerScience2008.pdf`, 2011

[4] ACM Java Task Force: Project Rationale. `http://jtf.acm.org/rationale/rationale.pdf`, 2011.

[5] Rossum G. van. Computer Programming for Everybody. `http://www.python.org/doc/essays/cp4e.html`

[6] European Higher Education Area Web Site. `http://www.ehea.info/, 2011`

[7] Ben-Ari M. Objects Never? Well, Hardly Ever! *Communications of the ACM*, **53** (2010), No 9, 32–35.

[8] Miller B., D. Ranum. Freedom to Succeed: a Three Course Introductory Sequence Using Python and Java. *Journal of Computing Sciences in Colleges*, **22** (October 2006), Issue 1.

[9] Pendergast M. Teaching Introductory Programming to IS Students: Java Problems and Pitfalls. *Journal of Information Technology Education*, **5** (2006), 491–515.

[10] Roy P. van, S. Haridi. Teaching Programming Broadly and Deeply: The Kernel Language Approach. In: Conference Proceedings of the Conference on Informatics Curricula, Teaching Methods and Best Practics (ICTEM 2002), July 10–12, 2002, Florianópolis,Brazil, 53–62.

[11] Radenski A. "Python First": A Lab-Based Digital Introduction to Computer science. In: ITICSE '06: Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, June 2006, Bologna, Italy, 197–201.

[12] Radenski A. Digital Support for Abductive Learning in Introductory Computing Courses. In: Proceedings of the 38th ACM Technical Symposium on Computer Science Education, SIGCSE '07, Covington, Kentucky, USA, March 7–10, 2007.

[13] Radenski A. Freedom of Choice as a Motivational Factor in Active Learning. In: Proceedings of the Fourteenth Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE 09, Université Pierre et Marie Curie, Paris, France, July 6–9, 2009, ACM Press.

[14] Radenski A. J. Furlong, V. Zanev. The Java 5 Generics Compromise Orthogonality to Keep Compatibility. *Journal of Systems and Software*, **81** (Sep. 2008), 2069–2078.

[15] Reges S. Back to Basics in CS1 and CS2. In: Proceedings of 37th SIGCSE Technical Symposium, Houston, Texas, 2006.

[16] Smith D., R. Cartwright. Java Type Inference Is Broken: Can We Fix It? In: Proceedings of OOPSLA'08 October 19–23, 2008, Nashville, Tennessee, USA.

[17] Schemers Inc. `http://schemers.com`, 2011

[18] Tiobe Programming Community Index, `http://www.tiobe.com`, 2011

[19] Mitchell W. A paradigm shift to OOP has occurred... implementation to follow. *Journal of Computing Sciences in Colleges*, **16** (2001), Issue 2, 94–105.

[20] Cooper S., W. Dann, R. Pausch. Teaching objects-first in introductory computer science. In: Proceedings of the 34th SIGCSE technical symposium on Computer science education. ACM New York, NY, USA, 2003, 191–195.

[21] Hu C. Rethinking of Teaching Objects-First. *Education and Information Technologies,* **9** (2004), No 3, Kluwer Academic Publishers, the Netherlands, 209–218.

[22] Horstmann C. Big Java: Compatible with Java 5, 6 and 7, 4/E, Wiley, 2009.

[23] Deitel P., H. Deitel. Java How to Program: Early Objects Version, 8/E, Prentice Hall, 2009.

[24] Lewis J. W. Loftus, Java Software Solutions: Foundations of Program Design, 6/E, Pearson, 2009.

[25] Reges S. Conservatively Radical Java in CS1. SIGCSE 2000. In: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, ACM New York, NY, USA, 2000, 85–89.

[26] Report. Introductory Computer Science Education at Carnegie Mellon University, `http://reports-archive.adm.cs.cmu.edu/anon/2010/CMU-CS-10-140.pdf`, 2011

[27] Rex P. Selling Haskell for CS1 in the USA. `http://www.cs.ou.edu/~rlpage/fnlCs1.pdf`, 2011

Atanas Radenski
Chapman University
Orange, CA 92866
e-mail: `radenski@chapman.edu`

Vladimir Zanev
Columbus State University
Columbus, GA 31907
e-mail: `zanev_vladimir@colstate.edu`

Received April 8, 2011
Final Accepted May 5, 2011