# Object-Oriented Programming

# 4

In this chapter you'll learn about the implementation of the JCoCo virtual machine while at the same time you'll be introduced to the Java and C++, two statically typed object-oriented programming languages. The primary focus of the chapter is on learning advanced object-oriented programming using Java and C++. Statically typed languages, like C++ and Java, differ from dynamically typed languages like Python in the way that type errors are caught. When running a Python program a type error can occur in any branch of code. One of the big problems with Python programming is that these errors may exist until every possible path in a Python program is executed. Testing Python code takes considerable effort to ensure every possible path is executed. While thorough testing is always a good idea, these type errors may not be discovered until much later in the development cycle.

Java and C++ programs are statically typed. This means that type errors are found at compile time, when the program is translated into executable format, without executing the program at all. Programmers must declare the types of all values or the compiler must be able to infer their type from the context of expressions in the program. With the declaration of value types in C++ and Java programs, the programmer is notified if any operation is not allowed without executing a single line of code. Run-time errors are still possible, but those run-time errors are due to logic problems and not due to type errors.

The JCoCo implementation will serve as nice examples while learning Java and C++. We'll compare Java and C++ when appropriate to show you the differences and similarities between the two languages. In the interest of seeing the big picture, we'll start with an overview of the JCoCo implementation as pictured in Fig. 4.1. JCoCo reads an input file which must be formatted according to the grammar specified in Sect. A.1. Two parts of the JCoCo implementation, the *scanner* and the *parser* are responsible for reading the input file. The scanner is implemented as a finite state machine. The parser is written as a top-down parser. The parser produces a list of *function* and *class* definitions which make up the *abstract syntax tree* definition of



K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_4

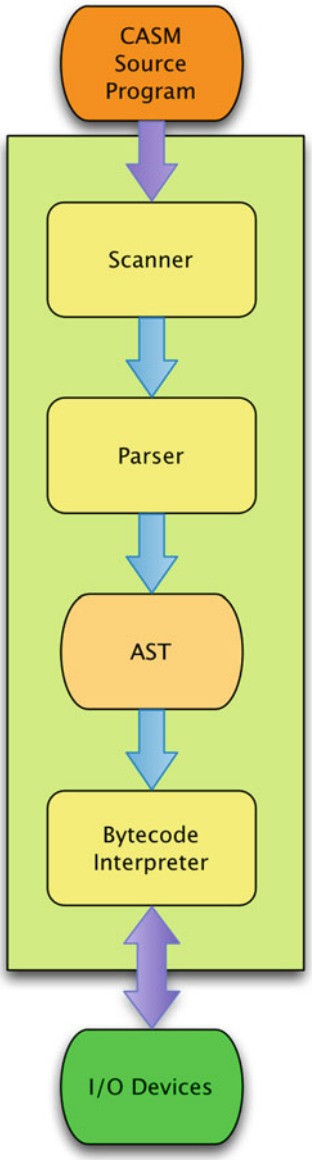


**Fig. 4.1** JCoCo

the program. The remainder of the JCoCo implementation makes up the bytecode interpreter.

The bytecode interpreter evaluates the abstract syntax tree (i.e. AST) which consists of *function* and *class* objects. The AST is interpreted within the context of *frame* objects. A frame is one activation record on the run-time stack of the executing pro-

gram. When the program begins, the interpreter starts execution at the *main* function, creating a frame for it and starting execution at the first instruction.

The examples of Java and C++ used within this chapter will come from the implementation of the scanner, parser, function, class, and frame classes along with classes for types and instances of types like integers, floats, strings, and lists.

JCoCo is written in Java. A similar project, CoCo, was earlier written in C++. JCoCo is an improved, more fully developed version of CoCo, rewritten in Java. JCoCo is a large project consisting of 56 source files and around 8,900 lines of code. Don't be intimidated, a lot of the code is repetitive. With such a large program, structuring it correctly is of the utmost importance. The JCoCo virtual machine is an interpreter of bytecode instructions somewhat like the Java Virtual Machine (i.e. JVM). The JCoCo interpreter reads assembly language source files called *CASM* files as you learned about in the last chapter.

Much of the design of CoCo and JCoCo is similar. JCoCo includes support for programmer-defined classes. CoCo does not support classes definition. Otherwise, the two implementations are very similar. The early part of this chapter will present

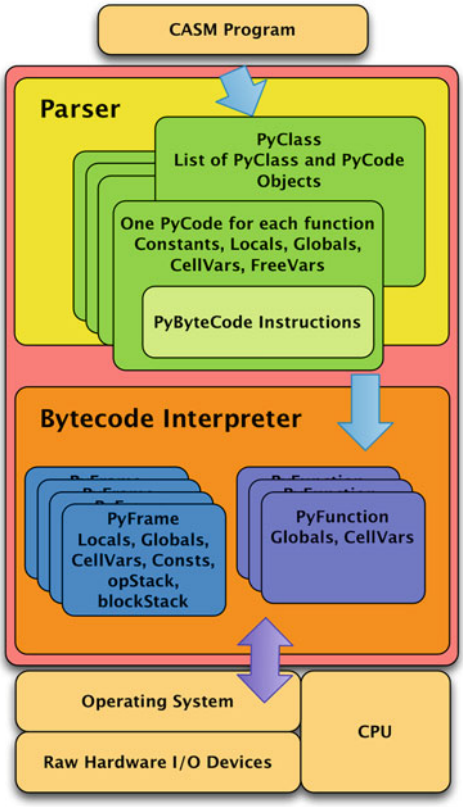


**Fig. 4.2** The JCoCo virtual machine

examples of both the Java and C++ implementations when appropriate to compare and contrast the two languages. Later sections will explore the details of the Java implementation of JCoCo.

Like other interpreters, the JCoCo/CoCo implementation is divided into some logical components: the scanner, parser, and bytecode interpreter. The *scanner* reads characters from the CASM source file and creates objects called tokens. The last chapter had many examples of CASM files. Tokens from a CASM file like the one in Sect. 3.2 and pictured in Fig. 4.2 include a *Function* keyword, a colon, a *main* identifier, a slash, an integer 0, another keyword *Constants*, another colon, a *None* keyword, and so on. These tokens are returned one at a time to the parser when the parser requests another token.

The *parser* reads the tokens one at a time from the scanner and uses them while parsing the source file according to the grammar for JCoCo given in *Appendix A*. The grammar given there is LL(1) so the parser is implemented as a recursive descent parser. Each non-terminal of the grammar is a function in the parser. The right hand sides of rules for each nonterminal defines the body of each function in the parser. There will be more on this later in the chapter. The result of parsing the source file is an Abstract Syntax Tree, or AST. This AST is an internal representation of the program to be interpreted.

The JCoCo bytecode interpreter is the part of the program, given the AST, that interprets the byte code instructions of each function. As the instructions are executed, the virtual machine interacts with I/O devices like the keyboard and the screen. Bytecode interpretation is the responsibility of several parts of the JCoCo implementation as you will read later. The last part of this chapter has a detailed explanation of the implementation of the JCoCo virtual machine.

## 4.1 The Java Environment

In Chap. 1 we learned that Java originated as part of the Green project. Today, Java is a robust language that contains many features that make it convenient for programmers while also being efficient and powerful. Java actually consists of two important tools: a Java Virtual Machine (i.e. JVM) and a Java compiler that compiles from Java source code to Java bytecode (which is what the JVM executes).

Let's consider the hello world example, written in Java. The Java program is given in Fig. 4.3. This program must be saved in a file called *HelloWorld.java*. The program is compiled and run using commands like this.

```
My Mac> javac HelloWorld.java
My Mac> java HelloWorld
Hello World
My Mac>
```

The *javac* program is the Java compiler which translates a Java *source program*, ending in an extension of *.java*, to a *bytecode file*, ending in *.class*. The bytecode file

```
public class HelloWorld {
 public static
  void main(String args[]) {
  System.out.println("Hello World");
 }
}
```

**Fig. 4.3** Hello world

is a binary file that is machine readable, but not human readable. The bytecode file is read by the *Java Virtual Machine* or *JVM* which is actually the program called *java* or *java.exe* if you are running on a Windows machine. The JVM interacts with the operating system to execute the bytecode file.

Typically, a Java program consists of many source code files which are compiled into many bytecode files. When programming in Java, each source code file must be named the same as the public class within the source code file. For instance, the code in Fig. 4.3 must be in a file called *HelloWorld.java* because the public class is called *HelloWorld*. Every Java source code file can have exactly one public class.

Writing a non-trivial Java program involves creating many classes and therefore many source files. When a Java project is compiled, the Java compiler looks at the dates of all source files and all bytecode files. Every time a file is created or modified its last modification date is changed. This is information that is maintained by every operating system including Mac OS X, Microsoft Windows, and Linux. If any bytecode file is found to be older than its corresponding source file, then the source file is recompiled to produce a new bytecode file with a newer date than its source code file. This mechanism of using timestamps to determine what needs to be recompiled is called a *make facility* for historical reasons which we'll revisit in the next section.

**Practice 4.1** Another program is written and compiled. Here is the error message from the compiler. What can you discern from the compile message? Why would this be important to Java?

```
test.java:1: error: class Test is public,
   should be declared in a file named Test.java
public class Test {
       ^
1 error
My Mac>
```

*You can check your answer(s) in Section* 4.34.1.

## 4.2 The C++ Environment

C++ and Java share a lot of syntax. C++ was designed first with Java starting development about 10 years later. As mentioned in chapter one, Bjarne Stroustrup was developing C++ during the early eighties. He designed the language to be backward compatible with C so there were some decisions already made for him like the need for *separate compilation* and the presence of a *macro processor*. C++ is one of the most widely used object-oriented languages today and continues to evolve. A standards committee now oversees C++ with regular revisions to the language like the C++11 revision which came out in 2011 and the 2014 version which contained small changes over the 2011 version. A new version was formally accepted late in 2017. Development of the C++ language is ongoing.

Using C/C++ for a programming project does not come without some risks. A significant problem, perhaps the most persistent problem over time, with C/C++ programs are memory leaks. C/C++ programmers must be disciplined in their allocation and deallocation of memory. It is common that programs that run for a long time will have a memory leak that has to be tracked down, which is a difficult task. In many languages a garbage collector takes care of freeing memory that is no longer needed by a program. A garbage collector cannot safely be included as part of C and C++ programs. Both C and C++ are designed to give the programmer maximum control. This means that more responsibility is left to the programmer and as a result programmers need to be very disciplined when using C/C++. There is more on this in Sect. 4.7.

C and C++ have many uses including operating system development, timing critical software, and detailed hardware access. Learning to program in C++ well will take you a long ways towards being a great programmer in any language. This chapter won't teach you everything you need to know to become a C++ programmer. That could be and is the topic of many books. But this chapter will introduce you to many of the important concepts and skills you'll need to become a good C++ programmer.

Like Java, C++ programs must be compiled before you can run the them. Java programs are compiled to Java bytecode and the bytecode is run on the JVM. C++ programs are compiled into the machine language of the CPU that will execute them. The operating system of the computer where a C++ program runs is responsible for loading the executable program and getting it ready to run but otherwise a compiled C++ program runs directly on the CPU of the machine for which it was compiled.

Figure 4.5 depicts the compilation process for C++ programs. Examine Fig. 4.4 to contrast that with the execution of Java programs. The C++ environment looks more complicated. But everything in the green box is actually accomplished using one compile command. Figure 4.6 contains the classic *hello world* program written in C++.

To run the hello world program it must first be compiled. Figure 4.5 shows the process of compiling a C++ program like the one that appears in Fig. 4.6. First, the macro processor reads the *iostream* header file and combines it with the rest of the source file. The *iostream* header file does not include the code for streams. It just declares the streams and the operators used to write to the *out* stream. The combined

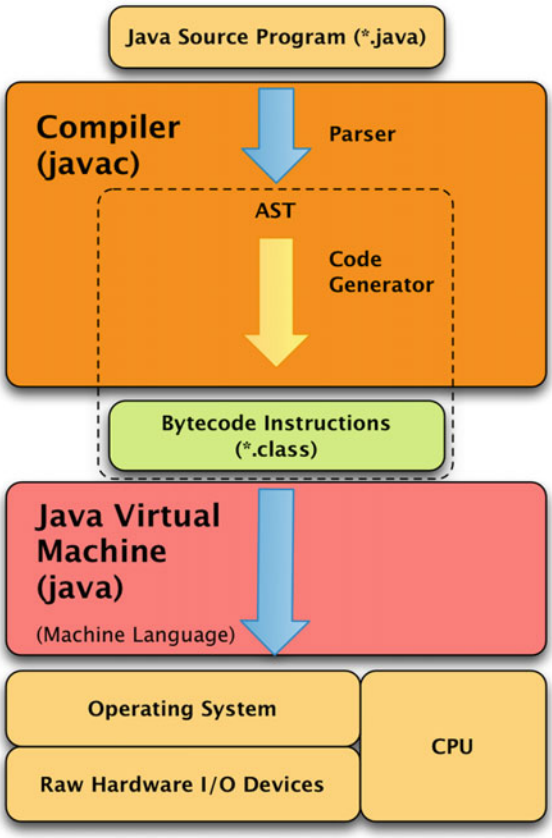


**Fig. 4.4** Java compiler and virtual machine

program text is sent to the compiler which parses the program and generates machine language code using an assembler. The machine language code is then linked with the *iostream* library to produce the executable code. Thankfully, this whole process is encapsulated in one command. There will be more about both the macro processor and I/O streams in the next sections.

Executing the $g$++ command compiles the program as shown in Fig. 4.7. By default $g$++ produces a program called a.out. To execute the program you type *a.out* and the operating system will load and run it. The default *a.out* can be renamed or a different name can be provided on the compile command.

The $-g$ option in Fig. 4.8 tells $g$++ to include debugging information in the program. The *-o* tells $g$++ to name the executable program *hello* instead of *a.out*.

To compile a C++ program you must have a C++ compiler installed on your system. The $g$++ compiler used in Fig. 4.7 is the GNU C++ compiler. This compiler is available for Mac OS X, Linux, and Microsoft Windows.

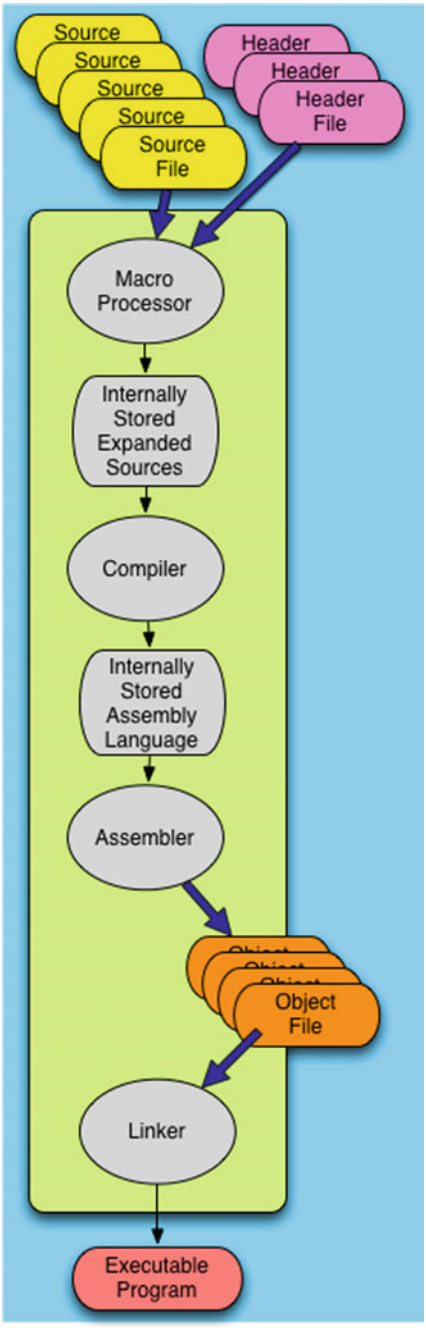


**Fig. 4.5** C++ compile

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
  cout << "Hello World!"<< endl;
}
```

**Fig. 4.6** hello.cpp

```
My Computer> g++ hello.cpp
My Computer> a.out
Hello World!
My Computer>
```

**Fig. 4.7** Compiling hello.cpp

```
My Computer> g++ -g -o hello hello.cpp
My Computer> hello
Hello World!
My Computer>
```

**Fig. 4.8** Include debug and name

### 4.2.1 The Macro Processor

The first line of the program in Fig. 4.6 is called a *macro processor directive*. The macro processor is a part of the C++ compiler that is responsible for pulling other files into the source program and sometimes for some simple editing of a source file to get it ready to be compiled. In this program the macro processor *includes* another file or library called *iostream*. The iostream file is called a header file because it defines functions and variables that exist in some other library or code on the system where it is compiled. Header files define the interfaces to these other libraries or code. When a header file is enclosed in angle braces, a less than/greater than pair, it is a system provided header file. More information on the macro processor and header files can be found in Sect. 4.9.1.

### 4.2.2 The Make Tool

A file system is the software and format that controls how files are stored on the hard disk of a computer. All operating systems have their own file systems and sometimes support multiple file systems. Microsoft Windows supports NTFS and Fat32 among others. Linux support ext2, ext3, reiserfs, and others. Mac OS X supports several file systems including HFS+. Every one of these file systems store attributes of every file including the date and time the file was last modified.

Make is a program that can be used to compile programs that are composed of modules and utilize separate compilation. C and C++ programs utilize separate compilation and typically you write a make file to compile programs written in these languages, or you use a tool to automatically create a make file. Java programs are not compiled via the make program because the make program is built into the Java compiler as was mentioned in the previous section.

The idea is simple. Every time a module is compiled by the C++ compiler it produces an object file. For instance, when *PyObject.cpp* is compiled, the C++ compiler writes a file called *PyObject.o*. For each of these files the date and time when it was last modified or created is stored with the file. After a compile the date on *PyObject.cpp* is older than the date on *PyObject.o*. When a programmer changes *PyObject.cpp*, its date will be newer than *PyObject.o*'s date.

Make uses this simple observation along with make rules to execute the compile commands necessary to make *PyObject.o*'s date newer than *PyObject.cpp*'s date. Here is a make rule for PyObject.cpp.

```
PyObject.o: PyObject.cpp PyObect.h
  g++ -g -c -std=c++0x PyObject.cpp
```

This rule says that *PyObject.o* must be newer than *PyObject.cpp* and *PyObject.h*. If either of these two files are newer then make will execute the command on the next line, which must be indented under the first line. The result of executing this compile command is to produce a new *PyObject.o* file with a newer date than either of the two source files.

To make the *coco* executable, all the object files must be linked together. To link everything together the first rule is written like this.

```
coco: main.o PyObject.o PyInt.o PyType.o ....
  g++ -o coco -std=c++0x main.o PyObject.o PyInt.o PyType.o ....
```

All 38 object files must be listed here. This says that the date on *coco*, the executable program, must be newer than the date on all its object files.

All these rules are placed in a file called *Makefile* in the same directory as the C++ source files. When *make* is invoked it will look for a file named *Makefile*. By keeping track of the dates, only the source files that have been updated will get recompiled and the *coco* executable will get recreated by linking together all the object files.

Writing a good *Makefile* is sometimes difficult and almost always error prone, so often there is a rule in the makefile called *clean*. Executing *make clean* will erase all object files so you can get a fresh compile. There are also tools like *autoreconf* that will generate a *Makefile* automatically with just a few inputs. Take a look at

the *rebuild* script in the CoCo distribution to see how this might be used. To use *autoreconf* you must have the automake tools installed on your system. But if you do, you can execute

```
./rebuild
./configure
make
```

to build the entire CoCo Virtual Machine. Without the automake tools you should still be able to execute the *configure* and *make* commands to build CoCo.

Separate compilation in C++ programs means that each module in the program is compiled separately. Each object file, generated by the compilation of a module, is produced independently of the other source files. This is important because large C++ projects often contain hundreds of C++ source files. Separate compilation means that only the small piece a programmer changes needs to be recompiled if the interface (i.e. the header file) to other modules does not change. After compiling the source files to object files, the object files can be linked together to form an executable program. Linking is a very fast operation compared to compiling.

**Practice 4.2** Using C++ there are no naming requirements for modules and classes like Java. So, when class *A* uses class *Test* both class *A* and class *Test* can be put into a file by any name. Why is this OK for C++ programs, but not for Java programs?
*You can check your answer(s) in Section* 4.34.2.

## 4.3 Namespaces

Line 2 of the program in Fig. 4.6 opens up the *std*, short for standard, namespace in the program. The first two lines of this C++ program are like importing a module in Python or a package in Java. When importing a module in Python the programmer writes an import statement like one of these two lines.

```
from iostream import * # merges the namespace with the current module
import iostream # preserves the namespace while importing the module
```

In Java the programmer would write an import statement somewhat like this, although not exactly.

```
import java.iostream.cout
```

The Python equivalent of a namespace is a module. Python modules can be imported in one of two ways, preserving the namespace or merging it with the existing namespace. In Java packages are the equivalent of a namespace and selected classes and objects can be imported from a package. Namespaces are important in C++, Python, and Java, because without them there would be many potential name conflicts between header files and modules that would create compile errors and prevent

```
#include <iostream>
int main(int argc, char* argv[]) {
  std::cout <<
     "Hello World!"<< std::endl;
}
```

**Fig. 4.9** Namespace std

programs from compiling, or in the case of Python - from running, that were otherwise correct programs. In C++ programs, if we didn't want to open the *std* namespace we could rewrite the program as shown in Fig. 4.9.

The safest way to program is to not open up namespaces or merge them together. But, that is also inconvenient since the whole name must be written each time. What is correct for your program depends on the program being written.

## 4.4 Dynamic Linking

Dynamic linking is related to namespaces, modules, and packages. Modern programming languages like C++ and Java are reliant on many libraries so programmers can solve problems instead of rewriting code that is common to more than one program. Libraries containing commonly used code are generally available to be used by programs written in a high level language, including C++ and Java programs. These libraries must be linked into your program to be able to use them. Figure 4.5 shows object files (i.e. modules) being linked together into a C++ program.

There is a problem with the picture in Fig. 4.5. Early C programs could be self-contained programs that relied on only a small number of system calls from the Unix operating system. However, modern C, C++, Java, and almost any other modern programming language are reliant on so many libraries that linking all of them together would be a problem in several ways.

- The size of the linked executable program would be huge taking up a lot of space in memory as it was executing.
- Any change in any library would require each program that uses it to be re-linked to get the new version of the library.
- There is no reason to have multiple copies of libraries, one for each program that uses it. This wastes space in addition to the overhead of having to manage multiple copies of libraries.

Modern languages don't *statically* link all the libraries that are required by a program. They *dynamically* link them. When you hear the word *dynamic* you should

think *run-time*. Libraries are generally linked at run-time. Software, often part of the operating system, detects when a library is going to be used by a program and loads it into memory and links it to the program that requests its services as the program is executing. Microsoft Windows calls these dynamically linked libraries *DLL's*. Windows includes services that let libraries be written so they can be dynamically linked to programs as they execute. Mac OS X and Linux also have the ability to dynamically link libraries. C++ programs often dynamically link to libraries that are provided by the C++ run-time libraries and other libraries that may be required by a program but have been supplied with the program.

Java programs also use dynamic linking. In fact, dynamic linking is built into the very foundations of Java. The JVM loads bytecode files (i.e. modules) as needed in your Java program. Java programs consist of *.class* files, called bytecode files, which must be in the current working directory or in a directory on the class path. The class path is a list of directories, or folders, where the JVM looks for bytecode files. The class path is recorded in an environment variable called *CLASSPATH*. Here is one example of a class path.

```
export CLASSPATH=./DBBrowser/lib/mysql-connector-
java-5.1.17-bin.jar:.:$CLASSPATH
```

The class path is a list of folders, or directories, where these dynamically loaded *.class* files can be found. Sometimes a whole group of classes are written to implement some library. For instance, this class path includes *mysql-connector-java-5.1.17-bin.jar*. This is actually what is called a *JAR* file. A JAR file stands for Java Archive, and is a zipped up set of *.class* files that are stored in compressed format. Dynamically linked libraries are so common to Java programs that JAR files were added as a means to conveniently group and redistribute collections of classes for Java programs. The bytecode files found in a JAR file are organized into packages. Importing something like

```
import java.io.BufferedInputStream
```

in a Java program would cause the *BufferedInputStream* class to be dynamically linked from the *java.io* package, which is a library provided by the Java run-time environment.

## 4.5 Defining the Main Function

Lines 3–5 of the hello world program in Fig. 4.6 define the *main* function. Every C++ program must have one main function, and only one. The main function should return an integer and it is given an integer and an array of character arrays which are the command-line arguments. The command-line arguments are elaborated on in more detail in the section on arrays and pointers later in this chapter.

Every Java program must also have a main function. However, when the program is run the class whose main function you want to run must be specified. In this way,

each class could potentially have a main function. The main of the specified class will be the first to run. The main function of the Java hello world program can be found in Fig. 4.3.

**Practice 4.3** Command-line arguments are typed in after the name of the program. For instance if a program is called *grep* then you might provide command-line arguments like this.

```
grep def *.py
```

Both C++ and Java programs can receive command-line arguments through the main function. With C++ the number of command-line arguments is passed as *argc* and the actual arguments are passed in the array of strings (i.e. character arrays) in the parameter named *argv* as declared in Fig. 4.9. The variable *argc* is always at least one for C++ programs but the length of the command-line arguments String array in Java may be zero if no command-line arguments are passed. Do you know why?
*You can check your answer(s) in Section* 4.34.3.

## 4.6 I/O Streams

Line 4 of the program in Fig. 4.6 prints *Hello World* to the screen. To be a little more precise, *cout* represents what is called a stream in C++. You can think of a C++ *stream* like a real stream with water in it. You can place things in the stream and they will be carried downstream. To place something in a C++ stream you use the << operator. Writing

```
cout << "Hello World";
```

places the string "Hello World" into the cout stream. This expression returns the cout stream. This means that multiple << operators can be chained together. Line 4 is like writing

```
(cout << "Hello World") << endl;
```

The parentheses are not needed in this example since the << is already left-associative. But they were included so you can see that the function call to << returns a stream which can be used in the next << operator to the right.

There are three streams automatically associated with programs. These three streams are associated with any program, whether C++, Python, Java, or other language. In C++, the first stream is called *cout* and by default it writes to the screen. The *cerr* stream also writes to the screen by default. The *cin* stream reads from the keyboard by default. The operator for reading from a stream is the right-shift operator, written *c*in>>*variable* where the variable will hold the value of its type which

was read from the stream. In each of these cases these streams can be redirected to read or write to different locations. Redirecting input and output is an operating system feature and not really associated with a specific programming language. You can search on the web for information about redirecting standard output, standard error, or standard input if you are interested in learning more about redirection.

Java programs have the same three streams. *System.out* is the name of the standard output stream. *System.err* is the standard error stream. *System.in* is the input stream. Figure 4.3 demonstrates writing to standard output in a Java program. The right-shift and left-shift operators are not used to read from and write to streams in Java. Instead the more traditional function call syntax is used.

## 4.7 Garbage Collection

Garbage collection occurs when dynamically allocated space needs to be returned to the pool of available space in memory. This space that is available for run-time allocation is called the *heap*. Every time an object is created a little of the heap memory of the computer must be reserved to hold that object's state information. When the object is no longer needed, the space on the heap has to be freed so it can be used by another object later.

Languages like Java and Python provide garbage collection as part of the underlying model of computation. They can do this because these languages are careful about how pointers are exposed to the programmer. In fact pointers are called references in these languages to distinguish them from pointers in languages like C and C++. The trade-off is that these languages take some control away from the programmer. Java, Python, and many languages that provide garbage collection require a virtual machine to execute their programs and the virtual machine takes care of managing and freeing unused memory.

Garbage collection can impact the run-time performance of a system. Languages like Java and Python aren't as well-suited to real-time applications where timing is critical. In these languages garbage collection can occur at any time. Usually, running of a program is not time critical and the time taken for garbage collection is negligible. The advantages of garbage collection typically far outweigh the possibility of memory leaks, but not in timing critical applications.

The existence of a run-time system that supports garbage collection, like the Java and Python virtual machines, means that those programs have less access to the underlying hardware of the machine. To safely free unused memory any garbage collection system must restrict the use of pointers in programs and as a result programs written in languages like Java and Python have less access to the details of the hardware platform. Again, this is not usually a problem for most programs, but there are instances where direct hardware access is important. Programs like operating systems are typically not written in Python or Java. To avoid any misconceptions, Android applications are written in Java, but the Android operating system itself is based on the Linux kernel which is implemented in C.

C++ programs must manage the allocation and freeing of heap space. But, it's not always clear when an object will no longer be used. A memory leak occurs when memory never gets freed even though the C++ program is done using it. There is extra work involved in writing C++ classes to insure that objects get freed when they are no longer needed. In the case of the CoCo virtual machine it is not safe to free objects once created because there is no reference counting in CoCo to decide when an object is no longer in use. Because objects are created and often referenced from multiple parts of a CASM program it is safe to simply free objects in CoCo. True garbage collection is needed in the C++ implementation of CoCo to make it a really useful virtual machine. As it stands, CoCo works for running short programs, but would not be suitable for long running applications.

For Java programs the garbage collector is a thread that runs once in a while and checks to see how many references are still referring to an object. If there are no parts of the program using an object, then it can be freed. Once in a while you might have a group of objects that are not being used, but all appear to be using each other. In this case the garbage collector can form a dependency graph and figure out that the objects involved form a cycle and no other objects outside the cycle are depending on the group of objects in the cycle. In this case, all objects in the cycle can be freed. The existence of a garbage collector greatly simplifies writing Java programs including JCoCo.

## 4.8 Threading

In the previous section it was mentioned that the JVM garbage collector runs in a different thread. A *thread* is a running sequence of instructions that shares access to objects with other threads. Each thread runs largely independently from other threads in the same program. You can think of each thread as essentially an independent program that is working with other threads in the same larger program to accomplish some work.

Java was built from the ground up to be a multi-threaded programming language. Every object in Java contains a lock that can be used to synchronize the behavior of multiple threads. When more than one thread is running, its work should not undo or change the work another thread is doing. When more than one thread is running there are two issues that need to be dealt with: synchronization of the threads, and communication between the threads.

Locks on objects make it possible for Java threads to both synchronize their work, and communicate with each other in structured ways. Every object in Java has a lock associated with it. There are also keywords in Java like *synchronized* that insure that only one method at a time may run on an object. This text won't teach you about Java threading, but it is an important topic and should be studied at some point.

C++ also has support for threads through the thread class in the standard namespace. However, C++ support of threads is quite a bit different than the Java support. For instance, C++ does not have keywords that allow for synchronized meth-

ods like Java. Threading in C++ requires a little more thought and work. C++ and Java are equally powerful in their support of multi-threaded programs, but given a choice, Java is the language to use for multi-threaded applications.

## 4.9 The PyToken Class

Object-Oriented programming is all about creating objects. Objects have *state information*, sometimes just called *state*, and methods that operate on that state, sometimes altering the state. If we alter the state of an object we call it a *mutable* object. If we cannot alter the object's state once it is created, the object is called *immutable*.

A class defines the state information maintained by an object and the methods that operate on that state. We'll start by examining the *PyToken* class in Fig. 4.10 since

```
1  package jcoco;
2  public class PyToken {
3
4      public enum TokenType {
5          PYIDENTIFIERTOKEN,
6          PYINTEGERTOKEN,
7          PYFLOATTOKEN,
8          PYSTRINGTOKEN,
9          PYKEYWORDTOKEN,
10         PYCOLONTOKEN,
11         PYCOMMATOKEN,
12         PYSLASHTOKEN,
13         PYLEFTPARENTOKEN,
14         PYRIGHTPARENTOKEN,
15         PYEOFTOKEN,
16         PYBADTOKEN
17     }
18
19     private String lexeme;
20     private TokenType type;
21     private int line;
22     private int col;
23
24     public PyToken(TokenType type, String lex, int line, int col) {
25         this.lexeme = lex;
26         this.type = type;
27         this.line = line;
28         this.col = col;
29     }
30
31     public TokenType getType() {
32         return this.type;
33     }
34
35     public String getLex() {
36         return this.lexeme;
37     }
38
39    // See getCol and getLine in the full source code.
40 }
```

**Fig. 4.10** The PyToken class

it is a simple immutable class. The PyToken class defines the token objects that are returned by the scanner to the parser during parsing of a JCoCo program. All the JCoCo code is in a package called *jcoco*. Line 1 declares that this class belongs in this *jcoco* package.

Lines 4–17 of Fig. 4.10 define the *TokenType* enum, which is short for *enumeration*. The *TokenType* enumeration defines the types of the tokens returned by the scanner. If you recall from Chap. 3, the syntax of CASM programs is pretty simple and these constant values are all the possible token types. Each constant serves as a name for each token type. With this enumeration it is possible to use these constant names in the Java code where the type of token is needed. For instance, here is one snippet of code from the JCoCo interpreter. Using descriptive constant names is useful in writing self-documenting code which you should always strive to do.

```
if (tok.getType() != TokenType.PYEOFTOKEN)
{
    badToken(tok, "Excpected End Of File (EOF)");
}
```

Lines 19–22 define the *instance variables* of the object (i.e. the object state). Each of these variables is declared private so that only the class' methods may access the state information directly. Lines 24–29 define the PyToken constructor which is called when a PyToken object is created. Here is how a PyToken object gets created.

```
PyToken t;
t = new PyToken(type, lex, line, column);
```

Of course, the variables *type*, *lex*, *line*, and *column* would all have to have values already and be of the proper types. For instance, the *lex* variable must be declared as a *String*. Java is a statically typed language, so all variables must be declared before they can be used. In this code the variable *t* was declared to have PyToken type.

There are a couple of things you can't see in the textbook. Because this class is defined in a package called *jcoco*, the result of compiling this class will be placed in a subdirectory named *jcoco*. Packages and subdirectories go together in Java organization of files. In addition, this file must be named *PyToken.java* since it contains the public class *PyToken*. This is also part of Java's organization of files.

### 4.9.1 The C++ PyToken Class

The implementation of *PyToken* in C++ looks a bit different than the Java version. Java and C++ both support separate compilation of code. Using Java each class is written in a separate file. Each file is compiled separately by the Java compiler. When to re-compile a Java class is decided based on dates of both the *.java* and the *.class* files.

In C++ there is no such make mechanism built into the compiler. Instead, the separate *make* tool provides this functionality as described in Sect. 4.2.2. In addition, the interface to the class (i.e. the declaration of the class' instance variables and methods) is separated from the actual code that implements the methods. So, the

```
1  #include <string>
2  using namespace std;
3  class PyToken {
4  public:
5     PyToken(int tokenType, string lex, int line, int col);
6     virtual ~PyToken();
7     string getLex() const;
8     int getType() const;
9     int getCol() const;
10    int getLine() const;
11 private:
12    string lexeme;
13    int tokenType;
14    int line;
15    int column;
16 };
17 const int PYIDENTIFIERTOKEN = 1;
18 const int PYINTEGERTOKEN = 2;
19 const int PYFLOATTOKEN = 3;
20 const int PYSTRINGTOKEN = 4;
21 const int PYKEYWORDTOKEN = 5;
22 const int PYCOLONTOKEN = 6;
23 const int PYCOMMATOKEN = 7;
24 const int PYSLASHTOKEN = 8;
25 const int PYLEFTPARENTOKEN = 9;
26 const int PYRIGHTPARENTOKEN = 10;
27 const int PYEOFTOKEN = 98;
28 const int PYBADTOKEN = 99;
```

**Fig. 4.11** The C++ PyToken class declaration - PyToken.h

*PyToken* class definition is separated into two files: the PyToken header file, named *PyToken.h*, and the method implementations located in *PyToken.cpp*. Figure 4.11 shows the contents of the header file declaration of the class. Only the *.cpp* source files are compiled using the compiler. The header files are included in the source files for use during compilation of the source files.

Much of the class declaration in the header file looks like the Java version. The *enum* defined in Fig. 4.10 is implemented as integer constants in Fig. 4.11 for no good reason. *Enums* are supported in C++ as well. Line 6 provides the declaration of a *destructor* which in general is used by C++ because C++ programs must free up their space since there is no garbage collector as there is in Java. However, the destructor in this case doesn't really have a purpose since these tokens don't have pointers to other objects. A destructor is needed precisely when an object contains pointers to other resources that must be freed. PyToken objects do not contain any pointers to other objects and hence the destructor has no purpose in this class.

The other difference worth noting is the use of *const* after the four methods that return values. This declares that these methods don't mutate the *PyToken* object. They only return values from the *PyToken* object. The use of *const* exists in C++ because C++ is very flexible in the way parameters are passed and values are returned. Declaring that a method is *const* helps C++ know where the method can be safely called and can optimize the performance of C++ programs.

```
1  #include "PyToken.h"
2  PyToken::PyToken(int tokenType, string lex, int line, int col) {
3     this->lexeme = lex;
4     this->tokenType = tokenType;
5     this->line = line;
6     this->column = col;
7  }
8  PyToken::~PyToken() {}
9  int PyToken::getType() const {
10    return tokenType;
11 }
12 string PyToken::getLex() const {
13    return lexeme;
14 }
15 // getLine and getCol omitted.
```

**Fig. 4.12** The C++ PyToken class implementation - PyToken.cpp

The C++ implementation of PyToken is given in Fig. 4.12. The first line includes the declaration of the *PyToken.h* header file. This is a macro processor directive to bring that source code into this file. By doing this, the *PyToken* class is declared for this source code.

The *PyToken::* that you see in Fig. 4.12 is a scope qualifier. It indicates that while none of this code is physically written inside the *PyToken* class definition, it is meant to be a part of the *PyToken* class.

Lines 3–6 of Fig. 4.12 use an arrow operator, written ->. In Java this is written as a dot. The arrow operator follows a pointer. In C++, *this* is a pointer that points to the current object. In Java, *this* is a reference and we use the dot notation to dereference the *this* reference. Pointers are the address of data in the memory of the computer. Pointers can be used in expressions to create new pointers using pointer arithmetic. In a programming language a pointer can point anywhere. A *reference* is much more controlled. References are somewhat like pointers except that they cannot be used in arithmetic expressions. They also don't directly point to locations in memory. When a reference is dereferenced using a dot, the run-time system does the lookup in a reference table.

This difference between references and pointers means that we can safely rely on every reference pointing to a real object where we don't necessarily know if a pointer is pointing to space that might be safely freed or not since the pointer might be the result of some pointer arithmetic. References are safe for garbage collection. Pointers are not.

## 4.10 Inheritance and Polymorphism

Object-oriented programming languages help us organize our code. One great advantage of organizing our code around objects occurs when we are able to re-use code.

```
1  #ifndef PYOBJECT_H_
2  #define PYOBJECT_H_
3
4  #include <string>
5  #include <unordered_map>
6  #include <vector>
7  #include <iostream>
8  using namespace std;
9
10 class PyType;
11
12 class PyObject {
13 public:
14    PyObject();
15    virtual ~PyObject();
16    virtual PyType* getType();
17    virtual string toString();
18    PyObject* callMethod(string name, vector<PyObject*>* args);
19
20 protected:
21     unordered_map<string, PyObject* (PyObject::*)(vector<PyObject*>*)> dict;
22     virtual PyObject* __str__(vector<PyObject*>* args);
23     virtual PyObject* __type__(vector<PyObject*>* args);
24     virtual PyObject* __hash__(vector<PyObject*>* args);
25     virtual PyObject* __repr__(vector<PyObject*>* args);
26 };
27
28 ostream& operator << (ostream& os, PyObject& t);
29 extern bool verbose;
30 #endif /* PYOBJECT_H_ */
```

**Fig. 4.13** The C++ PyObject header file - PyObject.h

Code re-use is important so that we can write something once and forget it while we solve other problems. But for code re-use to work we need a way of customizing this code for our purposes.

*Inheritance* is the mechanism we employ to re-use code in software we are currently writing. *Polymorphism* is the mechanism we employ to customize the behavior of code we have already written. In this section we'll look at some C++ code to see how inheritance and polymorphism are specified. In the next section we'll revisit the same code as implemented in Java.

Consider the header file for PyObject in Fig. 4.13. The CoCo and JCoCo virtual machines work on Python objects. Every data value in Python is an object, so this idea of objects is very pervasive in the JCoCo/CoCo implementations. In fact, it is so pervasive there are certain things that every object in Python must be able to do. Certain methods can be called on every Python object. To be able to re-use as much code as possible it makes sense to write that common code in one place. One such place is the *PyObject* class in the C++ CoCo implementation.

Every object in Python can be converted to a string. While the string representations vary, the mechanism to convert an object to a string is to call the *__str__* method on the object. This is declared on line 22 of Fig. 4.13. Since all objects should respond to this method, the *PyObject* class defines a toString method and a *__str__* which calls the toString method. The *__repr__* method is similar to the *__str__* method. In some case the two methods return exactly the same string.

But, the *__repr__* returns a string that if evaluated using the eval function, would construct the same object. For instance, consider this code.

```
x = [1,2,3]
y = eval(repr(x))
```

After evaluating this code, both *x* and *y* refer to lists of integers, where *y* is a complete copy of the contents of the list referred to by *x*.

Every object in Python responds to a number of basic method calls. CoCo does not attempt to implement all of them. But another that it does implement is the *__hash__* method which returns a hash value for all hashable objects in Python. This is used when the object is used as a key in a dictionary (i.e. hash table). Only immutable objects may be used as keys in dictionaries.

The *__type__* method returns the type of any Python object. Every Python object has a type. The type is returned via this method which is also an object.

There are a few things to note in Fig. 4.13 related to programming in C++. The first seven lines are called macro processor directives. Any line starting with a pound sign (i.e. #) is a macro processor directive. The first line is an *if-not-defined* directive and the second line is a *define* directive. The last line of Fig. 4.13 is an *endif* that goes with the first line. The pattern of *ifndef*, *define*, *endif* macro processor directives is needed because *include* directives often end up with circular references where include *A* includes *B* which may in turn includes *C* which includes *A* again. This kind of circular reference is avoided by defining *PYOBJECT_H_* in Fig. 4.13. Once the *PyObject.h* include is included, the *PYOBJECT_H_* is defined and if *PyObject.h* is included again through a circular reference, or even through another include including it without a circular reference, it will not get included twice. This pattern of *ifndef-define-endif* is used for every header file in C and C++ programming.

Line 10 declares a class (i.e. type) called *PyType*. This is called a forward declaration. The *PyType* class is used in this header file, but *PyType.h* also includes *PyObject.h* so the forward declaration was necessary because of the circular reference. Line 14 is a constructor for the PyObject class. Line 15 is the destructor declaration which again doesn't really do anything for this class.

The use of the keyword *virtual* is important. Virtual methods are methods that are included in the virtual function table of a C++ class. This virtual function table is how C++ implements *polymorphism*. When a virtual function is called, there is an extra lookup of the function's address because classes that inherit from this class may override any of the virtual functions. For instance, it can't be known at compile-time which version of *toString* should be called, the one in PyObject or one of the *toString* methods defined in a subclass of PyObject.

Examine the *PyObject::__str__* method shown in Fig. 4.14. This method calls *toString* and returns a new *PyStr* object as the result of converting the object to a string. The subtly here is that which toString will be called is unknown until this code actually executes. For instance, if the current object is a *PyInt* then the code would be executed from PyInt.cpp as shown in Fig. 4.15. But if a *PyList* were the current object, then the *toString* method would be executed from Fig. 4.16.

```
PyObject* PyObject::__str__(vector<PyObject*>* args) {
    ostringstream msg;

    if (args->size() != 0) {
        msg << "TypeError: expected 0 arguments, got " << args->size();
        throw new PyException(PYWRONGARGCOUNTEXCEPTION,msg.str());
    }

    return new PyStr(toString());
}
```

**Fig. 4.14** The CoCo str magic method

```
string PyInt::toString() {
    stringstream ss;
    ss << val;
    return ss.str();
}
```

**Fig. 4.15** The PyInt toString method

```
string PyList::toString() {
    ostringstream s;
    vector<PyObject*> args;
    s << "[";
    for (int i=0;i<data.size();i++) {
        s << *(data[i]->callMethod("__repr__",&args));
        if (i < data.size()-1)
            s << ", ";
    }
    s << "]";
    return s.str();
}
```

**Fig. 4.16** The PyList toString method

Both PyInt and PyList inherit from PyObject in the C++ implementation. So, polymorphism works because *toString* is declared virtual and therefore the determination of which *toString* to call is made through an extra lookup of the actual pointer to the function, in the virtual function table, at run-time.

Line 28 declares a function, which in this case is an overloaded left-shift operator (i.e. <<) that can be used to print objects. The implementation of this overloaded left-shift operator, from the file *PyObject.cpp*, relies on polymorphism to customize its behavior, like the *__str__* method also implemented in that module. The *toString* to get called will depend on which type of object this is called.

```
ostream& operator <<(ostream &os, PyObject &t) {
    return os << t.toString();
}
```

## 4.11 Interfaces and Adapters

In early Object-Oriented programming languages the specification of an interface was tied directly to a class. For instance, in the previous section we learned that *toString* was tied directly to the *PyObject* class and any classes that inherited from *PyObject*. This works great until you have some class that inherits from something other than *PyObject* and would like to use the polymorphism defined by the *toString* method. Then you have a situation where you would like to inherit from two different classes at the same time. C++ solves this problem with multiple inheritance. C++ classes can inherit from more than one class.

Java does a few things a little different than C++. First, unlike C++, every class in Java inherits from the *Object* class either directly or indirectly. There is one class hierarchy in Java of which every class participates. C++ has no built-in inheritance hierarchy. Using C++, a class that does not explicitly inherit from something does not inherit from anything. In Java a class that does not explicitly inherit from anything inherits from *Object*.

Secondly, multiple inheritance is not supported using Java, which simplifies inheritance and its implementation. But, Java solves the whole problem of interfaces being tied to class declarations by separating the two concepts. An interface is a promise to support certain methods in a class. Classes can implement as many interfaces as they wish, which is the Java way of achieving multiple inheritance. But, interfaces are in no way tied to the class hierarchy. What's more, you can declare a parameter to be of the type of an interface. Consider the code in Fig. 4.17.

Like the C++ version, the Java PyObject interface declares a *str* method, similar in purpose to the C++ *toString* method. Declaring the *str* method in the interface means that all classes that implement this interface must implement the *str* method.

While the interface declaration separates the interface from the class hierarchy, it also does not implement any of the code for the interface. It's often the case that many classes which implement an interface will have at least some common code. Either each class must implement the same code or the programmer may choose to

```
1  package jcoco;
2
3  import java.util.ArrayList;
4  import java.util.HashMap;
5
6  public interface PyObject {
7      public String str() ;
8      public PyType getType();
9
10     public void set(String key, PyObject value) ;
11     public PyObject get(String key) ;
12     public PyObject callMethod(String name, ArrayList<PyObject> args) ;
13 }
```

**Fig. 4.17** The PyObject interface

```
1  package jcoco;
2
3  public class PyObjectAdapter implements PyObject {
4      protected HashMap<String, PyObject> dict = new HashMap<String, PyObject>();
5      protected HashMap<String, PyObject> attrs = new HashMap<String, PyObject>();
6      protected String name;
7      protected PyType.PyTypeId type;
8
9      public PyObjectAdapter(String name, PyType.PyTypeId type) {
10         this();
11         this.name = name;
12         this.type = type;
13     }
14
15     public PyObjectAdapter() {
16         name = "PyObject()";
17         type = PyType.PyTypeId.PyClassType;
18         PyObjectAdapter self = this;
19         this.dict.put("__str__", new PyBaseCallable() {
20             @Override
21             public PyObject __call__(ArrayList<PyObject> args) {
22                 if (args.size() != 0) {
23                  throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
24                          "TypeError: expected 0 argument, got " + args.size());
25                 }
26
27                 return new PyStr(self.str());
28             }
29         });
30       ...
31     }
32
33     @Override
34     public PyType getType() {
35         return JCoCo.PyTypes.get(type);
36     }
37
38     @Override
39     public String str() {
40         return name;
41     }
42
43     @Override
44     public String toString() {
45         PyStr s = (PyStr)callMethod("__repr__",new ArrayList<PyObject>());
46          return s.str();
47     }
48
49     @Override
50     public void set(String key, PyObject value) {
51         this.dict.put(key, value);
52     }
53    ...
54 }
```

**Fig. 4.18** The PyObjectAdapter

use inheritance to write an adapter class that implements the interface and provides common code to several subclasses. This is the case in Fig. 4.18. A significant portion of the code is omitted for brevity here.

There are several things to note in this code. Lines 4–7 define protected variables. Protected variables are hidden (i.e. not accessible) from any code that is not in the same package, in this case the *jcoco* package. Line 10 of the code calls *this()* which

calls the default constructor on line 15 so that code common to both constructors need not be duplicated.

Line 33 uses a decorator called *@Override*. This decorator tells the compiler that the method that follows overrides or implements a method from a base class or interface. This is useful in case you make a spelling error or incorrectly specify a type of parameter of a method. Spelling a name wrong, or changing the type of a parameter will result in a method that will never get called because polymorphism only works when the name and the types of arguments all match. Otherwise you are just defining a different method since Java supports parametrically overloading names, meaning that two methods may have the same name if they have different types of parameters. So, *@Override* can be useful in catching mistakes that might otherwise be difficult to debug.

**Practice 4.4** We have seen how polymorphism is provided by the C++ and Java programming languages. Polymorphism is also provided by Python. Yet with Python we don't declare methods virtual, like C++, and we don't have an built-in class hierarchy like Java. How does polymorphism happen in Python? *You can check your answer(s) in Section* 4.34.4.

## 4.12 Functions as Values

Python is a dynamically typed language. As such, methods are looked up at run-time, not compile time. All methods and values in Python are objects that are stored in dictionaries within other objects so they can be looked up at run-time. The keys in these dictionaries are strings: the names of the values, methods, or functions.

To implement a virtual machine that works like Python's virtual machine, it is necessary to treat functions as values and store them in dictionaries, or hash tables, like Python's virtual machine implementation. C++ supports treating functions as values. Using C++ we can write the following.

```
dict["__str__"]=(PyObject* (PyObject::*)(vector<
PyObject*>*)) (&PyObject::__str__);
```

This code comes from the *PyObject* class in the C++ implementation of CoCo, in the file *PyObject.cpp*. There is a subtle nuance to this code. Once the *__str__* method is set in the object's dictionary any and all subclasses that inherit from *PyObject* and override the *__str__* method will automatically get the overridden method definition. There is no need to set *"__str__"* to point to the new, overridden *__str__* method. Because the *__str__* method is declared virtual, polymorphism means it only has to be set in the dictionary once.

SImilar code is not possible in Java because Java does not treat functions and methods as values. But there is a solution. A *class* can simulate a function or method.

```
1 public interface PyCallable extends PyObject {
2     public PyObject __call__(ArrayList<PyObject> args) ;
3 }
```

**Fig. 4.19** The PyCallable interface

In fact, Java contains support for doing just this. JCoCo implements its own version of run-time lookup of a method or function by using objects to simulate the functions and methods.

JCoCo defines an interface called *PyCallable*, shown in Fig. 4.19. This interface defines one method called *__call__*. This method takes a list of *PyObject* references and returns a *PyObject* as its result. This reflects the calling mechanism within Python. All Python functions are given a list of Python objects and return a Python object. This uniformity means that any class that implements the *PyCallable* interface can be called by calling its *__call__* method. This means that functions can be treated as values in JCoCo by encoding the functions as objects.

## 4.13 Anonymous Inner Classes

Interfaces in Java specify the methods that must be supported by classes that implement them. The *PyCallable* interface, described in the last section, specifies a *__call__* method. Like other interfaces, there are adapter classes that provide some common code for classes that choose to implement the *PyCallable* interface. The simplest of these is the *PyBaseCallable* class which is used in the functions implemented in the *PyObjectAdapter* class and the *PyCallableAdapter* class to avoid a circular reference problem within the object hierarchy. Another class also implements the *PyCallable* interface, named *PyCallableAdapter*, which is used by all other implementations of *PyCallable* other than those created in the *PyObjectAdapter* and *PyCallableAdapter* classes.

Figure 4.18 contains code on lines 19–29 that creates an instance of the PyBaseCallable adapter. This is an example of an anonymous inner class. Line 19 creates a class that has no name, but inherits from PyBaseCallable and overrides the *__call__* method. There is one instance of this *PyBaseCallable* class created, the instance for this *__str__* method. When an object wishes to call the *__str__* method on an object it calls the *callMethod* method of the *PyObjectAdapter* class.

The *callMethod* code in Fig. 4.20 looks up the method in the object's dictionary and if it finds it, it calls the *__call__* method on it, which in the case of the code on lines 19–29 of Fig. 4.18 is overridden to call the *str()* method on the object and return a *PyStr* object with the contents returned by the *str()* method.

Anonymous inner classes are very important in Java. An inner class is any class defined within another class. Inner classes are important because they provide a means to implement call-backs. When an event occurs in a Java program, like an event from a GUI program (i.e. a mouse-click) or a message being received from the

```
@Override
public PyObject callMethod(String name, ArrayList<PyObject> args) {
    PyCallable mbr = null;
    if (this.dict.containsKey(name)) {
        mbr = (PyCallable) this.dict.get(name);
        return mbr.__call__(args);
    }
    throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
        "TypeError: '" + this.getType().str() +
        "' object has no attribute '" + name + "'");
}
```

**Fig. 4.20** The *callMethod* code

internet, if a call-back has been registered to handle that event, then the call-back is called. Inner classes are the perfect way to implement a call-back because the inner class automatically has access to all the variables and methods of the outer object.

In the case of the code in Fig. 4.18, the inner class is anonymous, it does not have a name. This is okay because typically a call-back has one and only one instance created for it, for a particular outer object. Earlier versions of Java did not have anonymous classes leaving Java programs littered with inner classes that only ever had one instance created. Java programmers needed a more compact, precise syntax to manage call-backs and anonymous classes were introduced.

The advantage of the inner class defined on lines 19–29 of Fig. 4.18 is on line 27 where the *str()* method is called which is a member of *PyObjectAdapter* which is the class of the outer object in this instance. Since this is an inner class, we can directly call the *str()* method in this call-back. Anonymous inner classes are used extensively throughout the JCoCo implementation.

## 4.14 Type Casting and Generics

Line 21 of the C++ code in Fig. 4.13 is an example of declaring a variable *dict* whose type is defined by a template. A template is how C++ programmers write generic classes. A generic class is usually a container of some type, in this case a hash table. This hash table maps strings to functions that are given a *vector* of *PyObject* pointers and return a *PyObject* pointer. The *vector* class is again a template. The vector passed to these functions is a sequence of *PyObject* pointers.

Generics are an important part of object-oriented languages. Generics let programmers re-use classes, especially classes that are designed as data structures like maps and vectors. A *map* is a data structure mapping *keys* to *values*. The type of the keys and values can be practically anything. So, a generic *map* class provides the ability to map any type of *keys* to any type of *values*. In Java a *map* is called a *HashMap*. In C++ there are several kinds of map classes. Please note that in C++ the

```
private ArrayList BodyPart() {
        ArrayList instructions = new ArrayList();
        PyToken tok = this.in.getToken();
        this.target.clear();
        this.index = 0;
        if (!tok.getLex().equals("BEGIN")) {
            badToken(tok, "Expected a BEGIN keyword.");
        }
        ....
```

**Fig. 4.21** An ArrayList example

standard *map* class is not implemented as a hash table. It guarantees O(log n) insert and lookup time. A hash table guarantees an amortized complexity of O(1) insert and lookup. The *unordered_map* of C++11 is implemented as a hash table. Before C++11 this class was not included with C++.

Java has one type hierarchy. Everything inherits from Object either directly or indirectly. By having one type hierarchy the creators of Java could provide container classes for many of the data structures we need in our programs including *HashMap* and *ArrayList* which provides a means for storing a list of objects. For instance, if you needed a list of objects to be returned from a function you might code it as shown in Fig. 4.21.

The *BodyPart* function, a part of the *PyParser* module, returns a list of *PyByteCode* objects. When one of these *PyByteCode* objects is needed, we would be forced to write code like this to access the first PyByteCode object in the list.

```
ArrayList bp = BodyPart();
PyByteCode byteCode = (PyByteCode) bp.get(0);
```

The *(PyByteCode)*, with the parens, is called a *type cast* or just a *cast*. Casting is necessary when moving down in the inheritance hierarchy. A cast is a way to tell the Java compiler that you know the actual type of the value while the compiler does not. There is a run-time check that is inserted into your code. If the cast is invalid, the Java program will throw an exception so casting is safe. It's just not convenient and the extra run-time check is less than desirable, although arguably better than not checking at all.

Casting is the same in C++ and Java. The same issue occurs in C++. When moving down the inheritance hierarchy, a cast would be required. C++ has a datatype similar to Java's *ArrayList*, called *vector*. However, C++ does not have one super class of all other classes. So the *vector* datatype would be a little harder to write without something called generics.

Moving down the inheritance hierarchy in either Java or C++ requires the programmer to write more code. If we could avoid moving up the hierarchy in the first place, then moving down again would become unnecessary. This is the aim of generics. Generics were added to Java to make moving up and down the inheritance hierarchy, and therefore casting, unnecessary in many circumstances. Consider the real version of the *BodyPart* function in Fig. 4.22.

```
    private ArrayList<PyByteCode> BodyPart() {
        ArrayList<PyByteCode> instructions = new ArrayList<PyByteCode>();
        PyToken tok = this.in.getToken();
        this.target.clear();
        this.index = 0;
        if (!tok.getLex().equals("BEGIN")) {
            badToken(tok, "Expected a BEGIN keyword.");
        }
        ...
```

**Fig. 4.22** An ArrayList example using generics

```
template < class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator< pair<const Key,T> >
           > class unordered_map {
           ...
}
```

**Fig. 4.23** The unordered _map template

In this code the angle brackets (i.e. < and >) delimit the type of the *ArrayList*. The *ArrayList* is a list of *PyByteCode* elements. This declares the specific type contained in the *ArrayList* making the declaration of the *ArrayList* generic, so that it can be a container of any type, not just *Object* values. To declare the *ArrayList* class to be generic the Java creators would have changed its definition to look like this.

```
class ArrayList<T> {
    private T data[] = new T[10];
    ...
```

The type, $T$, becomes a parameter to the class declaration. A version of the class is created for each declared version of the ArrayList. So, when the *ArrayList <PyByteCode>* class is specified, a *PyByteCode ArrayList* object is created.

Python, since it is dynamically typed, does not need generics. Generics are only needed for statically typed languages like Java and C++. In C++ generics are called templates. A template is a parameterized class. Like Java, the parameter to the class is a type or types. Standard template containers in C++ include *unordered _map*, *map*, *vector*, *list*, *queue*, *stack*, *deque*, *set*, and *array* among others. Consider the declaration of the *unordered _map* template in C++ in Fig. 4.23. This template definition shows us that more than one type parameter can be used for a template or a generic in C++ or Java. In the case of the *unordered _map* there are five type parameters passed to the declaration of the map.

Diamond notation is one topic related to generics in Java. To save even more writing, Java programmers may use diamond notation when writing a generic object

declaration. The declaration of the variable *instructions* earlier in this section could have be written as follows if we would have used diamond notation.

```
ArrayList<PyByteCode> instructions = new ArrayList<>();
```

You can probably see the diamond in the code. Since *PyByteCode* is already written once on this line, using Java you don't have to write it again. The compiler can infer the type of the ArrayList as it is created from the type of the reference *instructions* pointing to it.

## 4.15 Auto-Boxing and Unboxing

In C++ you can declare a *vector* of *int* if you need to by writing

```
vector<int> intVec;
```

It is not possible to declare an *ArrayList* of *int* in Java. Templates in C++ can take any type as an argument to the class, even primitive types. In Java, only classes can serve as arguments to generics. The type *int* is a primitive type. That means it is not a class. However, the creators of Java understood this problem and provided wrapper classes for each of the primitive types so we could declare collections of *ints* for instance by wrapping each *int* as an *Integer*. So, while we can't declare an *ArrayList* of *int*, we can declare an *ArrayList* of *Integer* as follows.

```
ArrayList<Integer> intList = new ArrayList<>();
```

A wrapped *int* is created and added to our list as follows.

```
int x = 6;
Integer y = new Integer(x);
intList.add(y);
```

And getting it back out again involves writing some code like this.

```
x = intList.get(0).intValue();
```

This is the old way of wrapping and unwrapping integers, and other primitive types, in Java. Once again, programmers do this often enough they wanted a more compact way of wrapping and unwrapping primitive types. Java programmers refer to this as *boxing* and *unboxing*. Recent versions of Java support auto-boxing and unboxing. So, now in Java you can write the following.

```
int x = 6;
intList.add(x);
...
x = intList.get(0);
```

The variable *x's* value is auto-boxed when it is added to the list and auto-unboxed when it is extracted. Java determines when to box and unbox primitive values based on the type of value and the method being called. The syntax is much more compact and it is easier to read.

C++ does not support autoboxing and unboxing, but since you can declare template containers of primitive types it isn't as necessary to wrap and unwrap primitive values when using C++.

**Practice 4.5** The Java *ArrayList* contains two overloaded methods called *remove*. One takes an *int* parameter and removes an object at the specified index in the *ArrayList*. The other takes an *Object* as a parameter and removes the first instance of the object from the *ArrayList*. Why might this pose a problem?
*You can check your answer(s) in Section* 4.34.5.

## 4.16 Exception Handling in Java and C++

Java and C++ can throw exceptions and catch them as in many languages. Sometimes exceptions are thrown in code not written by us but code we use. For instance, indexing beyond the end of a vector. Other times we may wish to throw an exception. In C++ literally any type of value can be thrown.

Figure 4.24 shows how an object called a PyException is thrown using C++. This code was taken from the PyRange.cpp module. When *indexOf* is called beyond the end of a range object, CoCo throws a PyException object with a value of *stop iteration* as shown in Fig. 4.24.

Using Java, you throw values that inherit from the class Exception. Additionally, in some cases you must declare that a function throws an exception. For instance, in Fig. 4.25 the *indexOf* method declares that it throws *PyException*. It turns out in this case that declaring that the method throws this exception is optional because PyException inherits from *RuntimeException*. Exceptions that inherit from *RuntimeException* don't have to be declared to be thrown in Java.

Exceptions that are thrown can be caught and the C++ version of the exception is caught in PyFrame.cpp in the *FOR_ITER* instruction. The code for this appears

```
1  PyObject* PyRange::indexOf(int index) {
2      int val = start + index * increment;
3      if (increment > 0 && val >= stop) {
4          throw new PyException(PYSTOPITERATIONEXCEPTION,"Stop Iteration");
5      }
6      if (increment < 0 && val <= stop) {
7          throw new PyException(PYSTOPITERATIONEXCEPTION,"Stop Iteration");
8      }
9      return new PyInt(start + increment*index);
10 }
```

**Fig. 4.24** Throwing an exception in C++

```
    public PyObject indexOf(int index) throws PyException {
        int val = start + index * increment;
        if (increment > 0 && val >= stop) {
            throw new PyException(
            PyException.ExceptionType.PYSTOPITERATIONEXCEPTION,
            "Stop Iteration");
        }
        if (increment < 0 && val <= stop) {
            throw new PyException(
            PyException.ExceptionType.PYSTOPITERATIONEXCEPTION,
            "Stop Iteration");
        }
        return new PyInt(start + increment * index);
    }
```

**Fig. 4.25** Throwing an exception in Java

```
case FOR_ITER:
    u = safetyPop();
    args = new vector<PyObject*>();
    try {
        v = u->callMethod("__next__", args);
        opStack->push(u);
        opStack->push(v);
    } catch (PyException* ex) {
        if (ex->getExceptionType() == PYSTOPITERATIONEXCEPTION) {
            PC = operand;
        } else
            throw ex;
    }
    try {
        delete args;
    } catch (...) {
        cerr <<
         "Delete of FOR_ITER args caused an exception for some reason." <<
         endl;
    }
    break;
```

**Fig. 4.26** Catching an exception in C++

```
case FOR_ITER:
    u = this.safetyPop();
    args = new ArrayList<PyObject>();
    try {
        v = u.callMethod("__next__", args);
        this.opStack.push(u);
        this.opStack.push(v);
    } catch (PyException ex) {
        if (ex.getExceptionType() == ExceptionType.PYSTOPITERATIONEXCEPTION) {
            this.PC = operand;
        } else {
            throw ex;
        }
    }
    break;
```

**Fig. 4.27** Catching an exception in Java

in Fig. 4.26. To catch an exception it must be thrown in a *try block* or in some code called from a try block. Then the type of value caught in the *catch* must match the type of value thrown. Figure 4.27 provides the Java version of catching an exception. There is not much difference between C++ and Java in handling exceptions. The additional code in Fig. 4.26 on lines 14–20 are needed because C++ does not have garbage collection while Java does.

Figures 4.24 and 4.26 demonstrate how exception handling can be used to implement iteration within the CoCo interpreter, while Figs. 4.25 and 4.27 provide the Java version for JCoCo. When the end of an iteration is reached, a stop iteration exception is thrown and when caught it signals the end of the iteration.

```
1  void sigHandler(int signum) {
2    cerr << "\n\n";
3    cerr << "*******************************************************" << endl;
4    cerr << "            An Uncaught Exception Occurred" << endl;
5    cerr << "*******************************************************" << endl;
6    cerr << "Signal: ";
7    switch (signum) {
8        case SIGABRT:
9            cerr << "Program Execution Aborted" << endl;
10           break;
11       case SIGFPE:
12           cerr << "Arithmetic or Overflow Error" << endl;
13           break;
14       case SIGILL:
15           cerr << "Illegal Instruction in Virtual Machine" << endl;
16           break;
17       case SIGINT:
18           cerr << "Execution Interrupted" << endl;
19           break;
20       case SIGSEGV:
21           cerr << "Illegal Memory Access" << endl;
22           break;
23       case SIGTERM:
24           cerr << "Termination Requested" << endl;
25           break;
26   }
27   cerr << "-------------------------------------------------------" << endl;
28   cerr << "              The Exception's Traceback" << endl;
29   cerr << "-------------------------------------------------------" << endl;
30   for (int k=callStack.size()-1;k>=0;k--) {
31       cerr << "==========> At PC=" << (callStack[k]->getPC()-1) <<
32       " in this function. " << endl;
33       cerr << callStack[k]->getCode().prettyString("",true);
34   }
35   exit(0);
36 }
37
38 int main(int argc, char* argv[]) {
39   char* filename;
40
41   signal(SIGABRT,sigHandler);
42   signal(SIGFPE,sigHandler);
43   signal(SIGILL,sigHandler);
44   signal(SIGINT,sigHandler);
45   signal(SIGSEGV,sigHandler);
46   signal(SIGTERM,sigHandler);
47   ...
```

**Fig. 4.28** Signal handling

Exception handling is a means of handling conditions within a program, whether planned or unplanned. C++ and Java programs can throw and/or catch exceptions as needed. However some problems in C++, like division by zero errors, do not surface as exceptions. They are signaled instead which is the topic of the next section.

## 4.17 Signals

The C version of exception handling is signal handling. C programs can generate signals, but it is more common to put a signal handler in place to handle signals generated by the operating system. Figure 4.28 contains an excerpt of the code from main.cpp where a signal handler is implemented and is installed in main.

There are several types of signals and the code in Fig. 4.28 is written to catch all of the signals defined in the C standard. The constant signal types are defined in an include called *signal.h*. When a signal is generated the program immediately jumps to the signal handler passing it the signal value that was generated. The signal handler usually is written to report some type of error and then terminates. The signal handler presented in Fig. 4.28 does that. It prints a traceback of the program and then terminates.

## 4.18 JCoCo in Depth

The rest of the chapter will cover only the Java implementation of JCoCo. Many similarities exist to the C++ CoCo implementation. JCoCo is mostly a superset of the CoCo implementation. When there are differences between the two implementations they will be noted. Primarily JCoCo adds support for the creation of programmer-defined classes.

## 4.19 The Scanner

The JCoCo virtual machine reads a CASM file as depicted in Fig. 4.1. The virtual machine starts by using a scanner to return the tokens of the CASM file. It is common to implement a scanner as a finite state machine. The finite state machine consists of states and transitions between states depending on the characters read from the input file. The finite state machine accepts tokens of the CASM file. The finite state machine employed by JCoCo is depicted in Fig. 4.29.

When the parser is constructed, it first creates a scanner to read the tokens of the CASM program. The PyScanner's getToken method is written as a finite state machine to get the tokens of the CASM program. Figure 4.30 contains the *getToken*

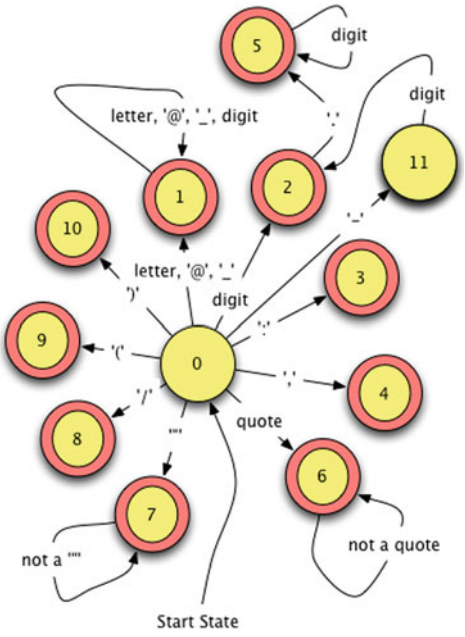


**Fig. 4.29** The JCoCo scanner FSM

and *putBackToken* methods for the scanner. The *putBackToken* method is capable of putting back one token which is used by the parser when it has to look ahead one token to determine its next action.

The scanner reads from a stream, which in the case of JCoCo is a *PushbackInputStream* so that a character can be *unread*. The scanner also keeps track of its position within the file so each token can carry along the position where it was found in the input file.

The start state is 0 as shown in the Fig. 4.29. There are several things to take note of in the finite state machine. First, identifiers are accepted by state 1 and are limited to letters and digits where the first character is a letter. Underscore characters and @ characters are considered letters by the scanner so tokens like @x_1 are recognized as identifiers by JCoCo even though that is an illegal identifier in Python. State 2 accepts integers. State 5 accepts floating point numbers which must have a decimal point. Scientific floating point notation is not accepted by JCoCo.

States 6 and 7 are responsible for recognizing strings. These states keep reading until a single or double quote is found to end the string. However, strings cannot have a quote or double quote in them as they are defined. For instance, the string 'how's it going?' is not allowed because there is no escape character implemented in JCoCo and the second quote would end the string. The entire implementation of the finite state machine can be found in PyScanner.java with only an excerpt of the code appearing in Fig. 4.30.

```
public PyToken getToken() {
    if (!this.needToken) {
        this.needToken = true;
        return this.lastToken;
    }

    try {
        next = this.in.read();
        if (next == -1) {
            type = TokenType.PYBADTOKEN;
            foundOne = true;
        }

        while (!foundOne) {
            this.colCount++;
            c = (char) next;
            switch (state) {
                case 0:
                    lex = "";
                    column = this.colCount;
                    line = this.lineCount;
                    if (isLetter(c)) state = 1;
                    else if (Character.isDigit(c)) state = 2;
                    else if (c == '-') state = 11;
                    ...
                    break;
                ...
            }
            if (!foundOne) {
                lex += c;
                next = this.in.read();
                if (next == -1) {
                    type = TokenType.PYEOFTOKEN;
                    foundOne = true;
                }
            }
        }
        this.in.unread((char)next);
        this.colCount--;
    } catch(IOException e) {
        System.err.println(e.getMessage());
    }
    PyToken t = new PyToken(type, lex, line, column);
    this.lastToken = t;
    return t;
}

public void putBackToken() {
    needToken = false;
}
```

**Fig. 4.30** PyScanner getToken and putBackToken methods

The input stream contains a method to put back the last character which is used by the scanner code on line 38 of the finite state machine loop in Fig. 4.30. The last character must be put back when a token is returned because the last character is not a part of that token. Consider state 1 for example. The finite state machine remains in state 1 as long as the character is still a letter or a digit. When it is neither, the *foundOne* variable is set to true, terminating the loop. But, that last character may be part of the next token and so is put back before returning.

Just before *getToken* returns a token, the token to be returned is saved. This is used by the *putBackToken* method. If the last token is put back then *needToken* is simply set to false. When *getToken* is called, lines 2–5 check to see if *needToken* is false and if so return the token that was put back by the *putBackToken* method. By saving the token before it is returned the last token is always remembered in case it needs to be returned again.

## 4.20 The Parser

The tokens of a CASM file are read by the parser and parsed according to the grammar rules in *Appendix A*. Each BNF non-terminal corresponds to one function in the parser. The parser returns an abstract syntax tree representing the CASM program. In this implementation, the abstract syntax tree is an *ArrayList* of PyCode and PyClass objects, which means the *ArrayList* is declared as a list of *PyObjects*. Figure 4.31 contains an outline of the *parse* method and some of the code called by *parse* which can be found in *PyParser.java*.

Each method of the parser corresponds to a non-terminal of the grammar. The implementation of each method method is determined by the right hand sides of its rules. The entire parser implementation is in *PyParser.java*. Figures 4.31 and 4.32 contain two excerpts of this code. Examining the rules for *ClassFunctionList*, *FunDef*, and *ConstPart* will shed some light on the implementations of the methods in Figs. 4.31 and 4.32.

```
CoCoAssemblyProg ::= ClassFunctionListPart EOF
ClassFunctionListPart ::= ClassFunDef ClassFunctionList
ClassFunctionList ::= ClassFunDef ClassFunctionList | <null>
ClassFunDef ::= ClassDef | FunDef
FunDef ::= Function colon Identifier slash Integer
    ClassFunctionList ConstPart LocalsPart FreeVarsPart
    CellVarsPart GlobalsPart BodyPart
ClassDef ::= Class colon Identifier [ ( Identifier ) ]
    BEGIN ClassFunctionList END
ConstPart ::= <null> | Constants colon ValueList
```

Starting with the *ClassFunctionList* non-terminal, its rules say that either it is empty (i.e. <null>) or it is a *ClassFunDef* followed by a *ClassFunctionList*. How do we know which rule to follow? The answer can be found by looking ahead one token. If we examine the *FunDef* rule, it must start with the keyword *Function* and that should be the next token to be read in the *ClassFunctionList* implementation unless

```
1  public ArrayList<PyObject> parse() {
2      try {
3          return PyAssemblyProg();
4      } catch (PyException e) {
5          this.in.putBackToken();
6          PyToken tok = this.in.getToken();
7          // print error message
8          System.exit(0);
9      }
10     // unreachable
11     return null;
12 }
13 private ArrayList<PyObject> PyAssemblyProg() {
14     ArrayList<PyObject> code = ClassFunctionListPart();
15     PyToken tok = this.in.getToken();
16     if (tok.getType() != TokenType.PYEOFTOKEN) {
17         badToken(tok, "Excpected End Of File (EOF)");
18     }
19     return code;
20 }
21 private ArrayList<PyObject> ClassFunctionListPart() {
22     PyObject obj = ClassFunDef();
23     ArrayList<PyObject> codeList = new ArrayList<PyObject>();
24     codeList.add(obj);
25     codeList = ClassFunctionList(codeList);
26     return codeList;
27 }
28 private ArrayList<PyObject> ClassFunctionList(ArrayList<PyObject> codeList) {
29     PyToken tok = this.in.getToken();
30     this.in.putBackToken();
31
32     PyObject obj = null;
33     String lexeme = tok.getLex();
34     if (lexeme.equals("Function") || lexeme.equals("Class")) {
35         obj = ClassFunDef();
36         codeList.add(obj);
37         codeList = ClassFunctionList(codeList);
38     }
39     return codeList;
40 }
41 private PyObject ClassFunDef() {
42     PyToken tok = this.in.getToken();
43     this.in.putBackToken();
44     PyObject obj = null;
45     if (tok.getLex().equals("Function")) {
46         obj = FunDef();
47     } else if (tok.getLex().equals("Class")) {
48         obj = ClassDef();
49     } else {
50         // throw  exception
51     }
52     return obj;
53 }
```

**Fig. 4.31** *PyParser.java* excerpt 1

the next part of the program is a class definition. In that case the *ClassDef* non-terminal requires a *Class* keyword. To determine what to do we get the next token in line 29 of Fig. 4.31, put it back right away, and check to see if it was a *Function* or *Class* keyword. If it was either of these, then the first rule is executed by calling *ClassFunDef* followed by *ClassFunctionList*. If *Function* or *Class* is not the next token, then we return the *ArrayList* passed to the method since we follow the <*null*> rule.

```
private PyCode FunDef() {
    PyToken tok = this.in.getToken();
    if (!tok.getLex().equals("Function")) {
        badToken(tok, "Expected Function keyword.");
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals(":")) {
        badToken(tok, "Excpected a ':'.");
    }
    PyToken funName = this.in.getToken();
    if (funName.getType() != TokenType.PYIDENTIFIERTOKEN) {
        badToken(funName, "Expected an identifier");
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals("/")) {
        badToken(tok, "Expected a '/'");
    }
    PyToken numArgsToken = this.in.getToken();
    int numArgs = 0;
    if (numArgsToken.getType() != TokenType.PYINTEGERTOKEN) {
        badToken(numArgsToken, "Expected an integer token");
    }
    try {
        numArgs = Integer.parseInt(numArgsToken.getLex());
    } catch (NumberFormatException e) {
        System.err.println(e.getMessage());
        System.exit(0);
    }
    ArrayList<PyObject> nestedClassFunctionList = new ArrayList<PyObject>();
    nestedClassFunctionList = ClassFunctionList(nestedClassFunctionList);
    ArrayList<PyObject> constants = ConstPart(nestedClassFunctionList);
    ArrayList<String> locals = LocalsPart();
    ArrayList<String> freevars = FreeVarsPart();
    ArrayList<String> cellvars = CellVarsPart();
    ArrayList<String> globals = GlobalsPart();
    ArrayList<PyByteCode> instructions = BodyPart();
    return new PyCode(funName.getLex(), nestedClassFunctionList, constants,
            locals, freevars, cellvars, globals, instructions, numArgs);
}
private ArrayList<PyObject> ConstPart(ArrayList<PyObject> nestedCFList) {
    ArrayList<PyObject> constants = new ArrayList<PyObject>();
    PyToken tok = this.in.getToken();
    if (!tok.getLex().equals("Constants")) {
        this.in.putBackToken();
        return constants;
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals(":")) {
        badToken(tok, "Expected a ':'.");
    }
    constants = ValueList(constants, nestedCFList);
    return constants;
}
```

**Fig. 4.32** *PyParser.java* excerpt 2

Why is an *ArrayList* of *PyObjects* passed to the *ClassFunctionList* method? This ArrayList is the abstract syntax tree "so far", as it has been read up to this point in the parser. The *ClassFunctionList* method adds to that ArrayList if it finds another function or class definition.

The *FunDef* method has only one rule to follow. It is responsible for building a PyCode object to return to the *ClassFunDef* method. When *FunDef* is called, we have already checked that the first token is the keyword *Function* so lines 2–5 could be omitted. The rest of the method gets tokens, checks them to see if they are the expected tokens, and calls other methods of the parser to read the rest of the function definition.

The *ConstPart* method has two rules to follow, like the *ClassFunctionList* method. Again, it must get a token to determine which rule to follow. If the next token is not *Constants*, then the empty rule is used and the *ConstPart* method returns an empty *ArrayList*. Otherwise, it returns an *ArrayList* of the constants used in the function. Each constant string is used to build a *PyObject* value for that constant. The ArrayList *nestedCFList* is passed to the *ConstPart* method because a nested class or function is itself a constant value stored in a *PyClass* or *PyCode* object respectively. When a constant like *code(g)* appears in the list of constants it tells the parser to look up the code for it in the list of nested classes or functions passed to the *ConstPart* method.

The code excerpts in Figs. 4.31 and 4.32 demonstrate that the functions of the parser are straightforward implementations of the rules in the grammar. Once in a while a lookahead token is needed to determine which rule to follow, but otherwise the parser gets tokens when required and calls other nonterminal methods when indicated by the rule. The trickiest part of writing the parser is probably determining what should be returned. This is dictated by the information that is required in the abstract syntax tree which is determined by the intended use of the information in the source file.

## 4.21 The Assembler

Before CoCo can execute the code in a function, all labels must be converted to target addresses in the instructions. Labels make no sense to the bytecode interpreter. Labels are convenient for programmers but are not for code execution. The assembly phase looks for labels and replaces any instruction jump label with the address to which it corresponds. For instance, consider the CASM program in Fig. 4.33. The *label00* identifies the instruction at offset 11 in the main function. The *label01* maps to offset 18 and *label02* maps to offset 19. The instructions on line 14, 17, and 23 need to get the offset, not the label, of their intended targets. This is the job of the assembler.

The assembler is simple enough to include in the parser code when the body part of a function is parsed. There are two parts to it utilizing a *HashMap* to remember and then update the target addresses in the code.

The code for the assembler is contained in two of the parser methods, the *LabeledInstruction* method and the *BodyPart* method. The grammar rules surrounding this code are provided here.

```
<BodyPart> ::= BEGIN <InstructionList> END
<InstructionList> ::= <null> | <LabeledInstruction> <InstructionList>
<LabeledInstruction> ::= Identifier colon <LabeledInstruction> |
                         <Instruction> | <OpInstruction>
<Instruction> ::= STOP_CODE | NOP | POP_TOP | ROT_TWO | ROT_THREE | ...
```

The code for *LabeledInstruction* adds each discovered label to a map from labels to integer offsets. Lines 32–39 of Fig. 4.34 do this when they discover an instruction contains a label. If the code finds a label, then line 34 adds the label to the map making it point to the offset, called *index* in the code.

Target locations are updated in the body of the function on lines 11–19 of Fig. 4.34. If an instruction is found that uses a label as its target, the instruction is deleted and a new instruction with identical opcode is created with the actual target address of the instruction.

## 4.22 ByteCode

A PyByteCode object is created for each instruction found in a CASM program. The class definition, partially defined in Fig. 4.35, shows an *enum* being declared with all possible opcodes. This *enum* construct in Java is convenient and powerful. An enum is actually a class definition that declares both the name of each enumerated value and any attributes associated with it. In the case of the *PyOpCode* enum each instruction

```
1  Function: main/0
2  Constants: None, "Enter a list: "
3  Locals: x, lst, b
4  Globals: input, split, print
5  BEGIN
6             LOAD_GLOBAL          0
7             LOAD_CONST           1
8             CALL_FUNCTION        1
9             STORE_FAST           0
10            LOAD_FAST            0
11            LOAD_ATTR            1
12            CALL_FUNCTION        0
13            STORE_FAST           1
14            SETUP_LOOP     label02
15            LOAD_FAST            1
16            GET_ITER
17 label00:   FOR_ITER       label01
18            STORE_FAST           2
19            LOAD_GLOBAL          2
20            LOAD_FAST            2
21            CALL_FUNCTION        1
22            POP_TOP
23            JUMP_ABSOLUTE  label00
24 label01:   POP_BLOCK
25 label02:   LOAD_CONST           0
26            RETURN_VALUE
27 END
```

**Fig. 4.33** listiter.casm

```
private ArrayList<PyByteCode> BodyPart() {
    ArrayList<PyByteCode> instructions = new ArrayList<PyByteCode>();
    PyToken tok = this.in.getToken();
    this.target.clear();
    this.index = 0;
    if (!tok.getLex().equals("BEGIN")) {
        badToken(tok, "Expected a BEGIN keyword.");
    }
    instructions = InstructionList(instructions);
    //find the target of any labels in the byte code instructions
    for (int i = 0; i < instructions.size(); i++) {
        PyByteCode inst = instructions.get(i);
        String label = inst.getLabel();
        if (!label.equals("")) {
            String op = inst.getOpCodeName();
            instructions.remove(instructions.get(i));
            instructions.add(i, new PyByteCode(op, target.get(label)));
        }
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals("END")) {
        badToken(tok, "Expected a END keyword.");
    }
    return instructions;
}

private PyByteCode LabeledInstruction() {
    PyToken tok1 = this.in.getToken();
    String tok1Lex = tok1.getLex();
    PyToken tok2 = this.in.getToken();
    String tok2Lex = tok2.getLex();
    if (tok2Lex.equals(":")) {
        if (!this.target.containsKey(tok1Lex)) {
            this.target.put(tok1Lex, this.index);
        } else {
            badToken(tok1, "Duplicate label found.");
        }
        return LabeledInstruction();
    }
  // code omitted here.
}
```

**Fig. 4.34** Assembling a program

name has associated with it the number of arguments that will be supplied with the instruction. Each instruction either has zero or one arguments, which appear immediately following the instruction in a CASM file. Referring back to Fig. 4.33 most instructions in that example have one argument with the exception of the *GET_ITER*, *POP_TOP*, and *POP_BLOCK* instructions which have zero arguments.

Enumerated values are convenient because they aid in writing self-documenting code. The enumerated values in the program are constructed as objects, one for each value enumerated in the declaration. They can be referred to by their enumerated value in code. For instance, in *PyFrame.java* a switch statement chooses between possible instruction values.

```
class PyByteCode {
    enum PyOpCode {
        BINARY_ADD (0),
        LOAD_CONST (1),
        COMPARE_OP (1),
        CALL_FUNCTION (1),
        ...;
        private int args;
        PyOpCode(int args) {
            this.args = args;
        }
        public int args() {
            return this.args;
        }
    };

    private static HashMap<String, PyOpCode> OpCodeMap = createOpCodeMap();
    private static HashMap<String, Integer> ArgMap = createArgMap();

    private static HashMap<String, PyOpCode> createOpCodeMap() {
        HashMap<String, PyOpCode> map = new HashMap<String, PyOpCode>();
        for (PyOpCode opcode : PyOpCode.values()) {
            map.put(opcode.name(), opcode);
        }
        return map;
    }

    private static HashMap<String, Integer> createArgMap() {
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        for (PyOpCode opcode : PyOpCode.values()) {
            map.put(opcode.name(), opcode.args());
        }
        return map;
    }
    // code omitted here.
    public PyByteCode(String opcode, int operand)  {
        if (!OpCodeMap.containsKey(opcode)) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
                                  "Unknown opcode "+opcode);
        }

        this.opcode = OpCodeMap.get(opcode);
        this.operand = operand;
        this.label = "";
    }
    // code omitted here.
}
```

**Fig. 4.35** Static initialization

```
inst = this.code.getInstructions().get(this.PC);
switch (inst.getOpCode()) {
    case LOAD_FAST:
        u = this.locals.get(this.code.getLocals().get(operand));
        if (u == null) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
                    "NameError: name '" + this.code.getLocals().get(operand) ...
        }
        this.opStack.push(u);
        break;
    ...
```

This code gets the opcode from the next instruction. The switch statement is written with each instruction opcode enumerated in the case statements. This makes the code very clear when examining it. The behavior of the instructions is associated with the name of each instruction.

To build the PyByteCode objects from the CASM file it is necessary to translate the string opcodes, like *"LOAD_FAST"* into their actual opcodes, like *LOAD_FAST*. To accomplish this there has to be a way to look up a string and find its corresponding opcode. This lookup is done in O(1) time by using a *HashMap*. The hash map is created once, when the program begins. When code is executed once and only once at program initialization, it is called *static initialization*. Java and C++ both support static initialization of values.

Internally to the PyByteCode class there are two statically allocated maps that help in translating each instruction that is read by the parser into a PyByteCode object. The code in Fig. 4.35 appears in the *PyByteCode* module. The two variables *OpCodeMap* and *ArgMap* are statically initialized and available to all code implemented in the class. *OpCodeMap* is used when an opcode name is found in a CASM file. It serves to verify it is a valid instruction and to provide a translation to its enumerated value. *ArgMap* provides a count of the number of operands, either 0 or 1, allowed for the instruction. For example, looking up *"BINARY_ADD"* as *OpCodeMap["BINARY_ADD"]* would yield the enumerated value *BINARY_ADD*.

Static initialization of variables can be helpful when you have one-time code that needs to be run during program initialization. This section demonstrates how to do this using Java. Similar code exists for C++. See the module *PyByteCode.cpp* and *PyByteCode.h* in the CoCo implementation for an example of doing this using C++ for further details. In this Java version of it the two functions that create the maps are executed when called by the static initialization on lines 17 and 18 of the code in Fig. 4.35.

## 4.23 JCoCo's Class and Interface Type Hierarchy

The JCoCo implementation consists of approximately fifty six classes and interfaces. Approximately fifty six classes because JCoCo continues to grow and evolve. Class inheritance is used for code re-use and polymorphism throughout the implementation. Figure 4.36 provides a look at the hierarchy of classes and interfaces. The PyBuiltIns represent a collection of classes for all the built-in functions provided with JCoCo which include concat, print, fprint, tprint, iter, len, open, and repr. The fprint and tprint built-in functions are not part of Python. The fprint function is a functional version of print that takes one value and returns the fprint instance. The tprint built-in function takes a tuple and prints the elements of the tuple with spaces separating values in the tuple. PyBuildClass is similar to a built-in function, but is only accessible via the *LOAD_BUILD_CLASS* instruction.

PyIterators represents the collection of all iterator classes which include all the iterable values supported by JCoCo including dictionaries, lists, files, funlists (which are functional lists implemented as head/tail links). The key shows that the dark grey classes are used internally in the JCoCo implementation and are not available to CASM programs. These classes are part of the internal implementation of JCoCo and are not accessible to the programmer. Some of them have been described earlier

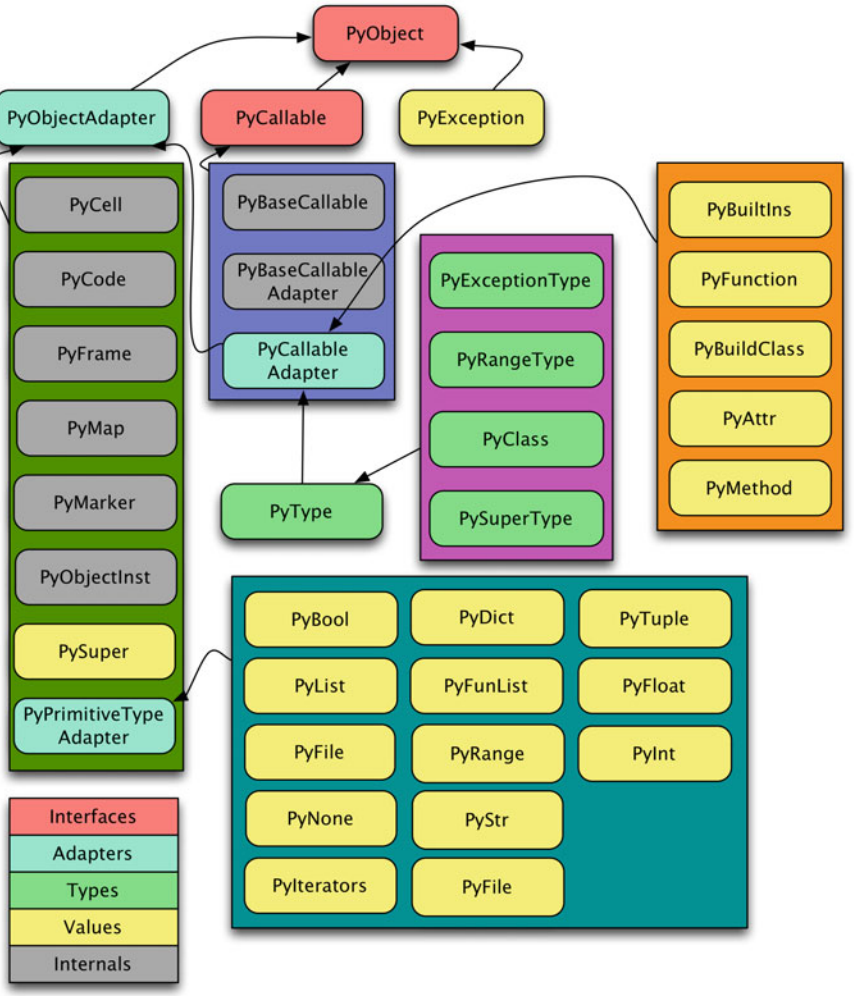


**Fig. 4.36** JCoCo type hierarchy

in this chapter. The PyType class is used by all but two of the JCoCo types of values in the construction of their type objects. The type of exception and range objects had to inherit from the PyType class so the behavior of calling the type could be overridden since these two types can be called to build either an exception object or range object, respectively.

There are two interfaces and three adapter classes. When possible, adapters were written to allow code to be shared between multiple classes. Consider the PyPrimitiveTypeAdapter class. This class defines the magic methods *__repr__*, *__str__*, *__hash__*, *__iter__*, and *__type__*. These methods are called by the *repr*, *str*, *hash*, *iter*, and *type* methods.

The PyException class is one interesting example of the need for multiple inheritance in Java. PyException inherits from RuntimeException. By inheriting from RuntimeException, JCoCo exceptions don't have to be declared to be thrown in each and every method that either throws or calls something that could throw an exception. Without inheriting from RuntimeException pretty much every method in JCoCo would have to be declared as possibly throwing a PyException which would have made quite a mess of the code.

Since PyException inherits from RuntimeException it cannot inherit from PyObjectAdapter. Instead, PyException implements the PyObject interface and therefore must re-implement the methods common to the PyObjectAdapter class. With multiple inheritance this could have been avoided. But, this is the only case where multiple inheritance would have been useful in this collection of classes.

JCoCo supports a number of different types of values, including integers, floats, dictionaries, lists, strings, booleans, and a few others. Each of these values has a type associated with it. Each JCoCo object has a method called *getType* that returns its type. It turns out even *types* are objects and they too have a type. Calling *getType* on a type returns the type named *type*. This has to end somewhere and it does with the type object named *type*. The type of *type* is *type*. This comes from the first two lines of the *initTypes* function in the JCoCo.java source file. The type object is created on the first line and is created with itself as its own type identifier.

```
PyType typeType = new PyType("type", PyTypeId.PyTypeType);
PyTypes.put(PyTypeId.PyTypeType, typeType);
```

The next few sections will dive deeper into a few of the JCoCo classes and interfaces to explore their purpose and how they fit in to the larger implementation of JCoCo.

**Practice 4.6** The existence of a class named *PySuper* suggests that classes can be built dynamically (i.e. at run-time). The need for PySuper stems from needing to look up the superclass dynamically, while the program is running, because in general it cannot be known before its use. What instruction in appendix A is responsible for getting JCoCo ready to dynamically create a class?
*You can check your answer(s) in Section* 4.34.6.

## 4.24 Code

A CASM function consists of more than just a sequence of PyByteCode objects. There is the name of the function, the number of arguments passed to the function, the list of constants used by the function, the local variables, the global variable references, and any enclosed functions or classes declared within this CASM function. All this information and more is encapsulated within a PyCode object.

```
package jcoco;

import java.util.ArrayList;
import jcoco.PyException.ExceptionType;
import jcoco.PyType.PyTypeId;

class PyCode  extends PyObjectAdapter {
    private String name;
    private ArrayList<PyObject> nestedClassFunctions;
    private ArrayList<String> locals;
    private ArrayList<String> freevars;
    private ArrayList<String> cellvars;
    private ArrayList<String> globals;
    private ArrayList<PyObject> consts;
    private ArrayList<PyByteCode> instructions;
    private int argCount;
    ... // methods and constructors omitted.
}
```

**Fig. 4.37** The PyCode class instance variables

From a Java programming perspective, this code not that unique. It is a container for all the information that goes along with each function. The class declaration of instance variables is given in Fig. 4.37.

PyCode objects cannot be executed. There is no way to *run* a PyCode object. To run code you need two things: the code and the environment in which it should be run. The environment is the variables, functions, and other values that are already defined and initialized before the function executes. The environment and code are both provided to PyFunction objects when they are executed, which is described in more detail in Sect. 4.25.

When a Python function is encoded as a PyCode object there are two lists that are necessary. They reflect the eventual contents of the environment. The free variables are variables that are referred to that exist in the environment and not in the code of the function. The other list, the cellvars, are a list of variables that come from the environment or are part of an inner function's environment that may be modified and therefore have to be indirectly referenced. This means that we go through an extra step to reference cellvars so we can update their values while accessing them from another environment. See Sect. 4.25 for an example.

## 4.25 Functions

Each function in a CASM file is scanned by the parser and a PyCode object is created in the abstract syntax tree to represent the code, its name and number of arguments along with its declaration of constants, locals, freevars, cellvars, and globals

```
1  public PyFunction(PyCode theCode, HashMap<String, PyObject> theGlobals,
2          PyObject env) {
3      PyTuple tuple = (PyTuple)env;
4      this.cellvars = new HashMap<String, PyCell>();
5      this.code = theCode;
6      this.globals = theGlobals;
7
8      for (int i = 0; i < theCode.getFreeVars().size(); i++) {
9          this.cellvars.put(theCode.getFreeVars().get(i),
10                           (PyCell)tuple.getVal(i));
11     }
12
13     PyFunction self = this;
14     this.dict.put("__call__", new PyCallableAdapter() {
15         @Override
16         public PyObject __call__(ArrayList<PyObject> args)  {
17             return self.__call__(args);
18         }
19     });
20 }
21 @Override
22 public PyObject __call__(ArrayList<PyObject> args)  {
23     if (args.size() != this.code.getArgCount()) {
24         throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
25                         "Type Error: expected "+this.code.getArgCount() +
26                         " arguments, got "+args.size());
27     }
28     PyFrame frame = new PyFrame(this.code, args, this.globals,
29                           this.code.getConsts(), this.cellvars);
30     PyObject result = frame.execute();
31     return result;
32 }
```

**Fig. 4.38** The PyFunction constructor and __call__ method

as described in Sect. 4.24. But, PyCode objects are not callable as shown in Fig. 4.36. To be callable you need both the code and an environment in which to execute the code. The environment fills in the gaps so to speak. The freevars are not defined within the function's code. The freevars come from the environment.

The code in Fig. 4.38 provides the constructor and the call method for the PyFunction class. The constructor builds what is called a closure from the environment and the code. The closure is initialized on line 8–10 where we iterate over the free variables in the code mapping the cell variables in the closure from their free variable names to their cells variable values. All variables that are accessed from the closure are accessed indirectly, through cell variables.

When a function gets called, the *__call__* method gets called. When this occurs a new PyFrame object is created. The PyFrame contains the program counter and space for local variables to be stored. A PyFrame object is executed by calling the execute method.

**Practice 4.7** What are the free variables and bound variables in this Python function?

```
def f(x):
    y = x
    return aVal + lstInts[0] + y
```

*You can check your answer(s) in Section* 4.34.7.

## 4.26 Classes

User-defined classes in JCoCo are collections of PyFunction objects and nested classes. The class contains the name of the class, (i.e. its type) its super class, and a list of PyFunction or PyClass objects as shown in Fig. 4.39. This reflects the implementation in Python. For instance, while it's not normally written this way, to add two integers together it is possible to write this.

```
z = int.__add__(x,y)
```

This code looks up the addition magic method in the int class and calls it passing the two arguments to the function. While addition is a method called on an integer object, it can also be called on the class by providing both integers.

```
> python3.2
Python 3.2.5 (v3.2.5:cef745775b65, May 13 2013, 13:37:00)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int.__add__(4,6)
10
>>>
```

When a PyClass is constructed it is passed a list of inner classes and one PyCode object for each of its methods. The PyCode objects are used to build PyFunction objects on lines 22–23. The PyFunction objects are placed in the *attr* dictionary. The variable *attr* of the class holds attributes that are to be passed on to any instance of this class. The class contains the functions that will become methods of any instance of the class. For example, the *__add__* function in the int class will become a method in an instance of the int class.

## 4.27 Methods

Instances of a class are created by calling their class. For instance, writing *Dog("Mesa")* creates an instance of the *Dog* class. The code that is executed when a class is called is shown in Fig. 4.39 on lines 48–54, which calls the *initInstance* method on lines 36–47. In this code all PyFunctions found in the *attrs* variable are

```
1  public PyClass(String name, ArrayList<PyObject> nestedclassesandfuns,
2              String baseClass, HashMap<String, PyObject> globals)  {
3      super(name, PyTypeId.PyClassType);
4      this.baseClass = baseClass;
5      this.name = name;
6      this.classesandfuns = nestedclassesandfuns;
7      this.globals = (HashMap<String, PyObject>)globals;
8      this.attrs.put("__name__", new PyStr(name));
9      for (int i = 0; i < classesandfuns.size(); i++) {
10         if (classesandfuns.get(i).getType().typeId() == PyTypeId.PyCodeType) {
11             PyCode code = (PyCode) classesandfuns.get(i);
12             ArrayList<PyObject> env = new ArrayList<PyObject>();
13             for (int j = 0; j<code.getFreeVars().size(); j++) {
14                 String freeVar = code.getFreeVars().get(j);
15                 if (freeVar.equals("__class__")) {
16                     env.add(new PyCell(this));
17                 } else {
18                     throw new PyException(ExceptionType.PYMATCHEXCEPTION,
19                      "Error: Found unexpected freevar in class declaration.");
20                 }
21             }
22             PyFunction fun = new PyFunction((PyCode) classesandfuns.get(i),
23                 globals, new PyTuple(env));
24             this.attrs.put(fun.callName(), fun);
25         } else if (classesandfuns.get(i).getType().typeId() ==
26                    PyTypeId.PyTypeType) {
27             PyClass cls = (PyClass) classesandfuns.get(i);
28             this.attrs.put(cls.getName(), cls);
29         } else {
30             throw new PyException(ExceptionType.PYMATCHEXCEPTION,
31                         "TypeError: expected a Function or Class, got "+
32                         classesandfuns.get(i).getType().str());
33         }
34     }
35 }
36 public void initInstance(PyObjectAdapter obj)  {
37     if (!baseClass.equals("")) {
38         ((PyClass)globals.get(baseClass)).initInstance(obj);
39     }
40     for (String name : this.attrs.keySet()) {
41         if (this.attrs.get(name).getType().typeId() ==
42               PyTypeId.PyFunctionType) {
43             obj.dict.put(name, new PyMethod(name, obj,
44                 (PyCallable)this.attrs.get(name)));
45         }
46     }
47 }
48 @Override
49 public PyObject __call__(ArrayList<PyObject> args)  {
50     PyObjectAdapter obj = new PyObjectInst(this);
51     initInstance(obj);
52     ((PyMethod) obj.dict.get("__init__")).__call__(args);
53     return obj;
54 }
```

**Fig. 4.39** The PyClass constructor and __call__ method

passed on to the instance of the class (i.e. the object instance) as PyMethod objects. Then, on line 52 the object's constructor is called to perform any initialization of the object. Continuing our example from the previous section, this means that like in Python, the following code can be disassembled and executed in JCoCo.

```
1  @Override
2  public PyObject __call__(ArrayList<PyObject> args)  {
3      args.add(this.self);
4      PyObject result = fun.__call__(args);
5      //take self back out of args because when a method
6      //is called no one would suspect that a side-effect
7      //was that args was mutated.
8      args.remove(args.size()-1);
9      return result;
10 }
```

**Fig. 4.40** The PyMethod __call__ method

```
> python3.2
Python 3.2.5 (v3.2.5:cef745775b65, May 13 2013, 13:37:00)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int(4).__add__(6)
10
>>> 4 + 6
10
>>>
```

The integer 4 had to be written *int(4)* because otherwise it would look like there was a decimal point in the number. Syntactically it would not be an integer in that case. Of course, the more usual way to call this method is by using the overloaded + operator.

The PyMethod class is a wrapper class for a PyFunction. A PyFunction of a class is passed on to any instance of that class as a PyMethod as shown in Fig. 4.39. Calling a method on an object is usually written as *object.method(arg1, arg2, ...)*. It is useful to know that JCoCo passes the list of args in reverse order. So the last argument is first, followed by the second to last, and so on.

When a method is called as in *object.method(args)* the first argument passed to the method is always *self* in Python. This *self* variable is the reference to the *object*. Since the arguments are passed in reverse order in JCoCo, the *self* argument can be added to the end of the arguments passed to the function that the method encapsulates. This is shown in Fig. 4.40 on line 3 where the reference to the current object is added to the end of the ArrayList of arguments. Note that line 8 removes self from the *args* ArrayList before returning so the calling code won't see a modified list of arguments.

**Practice 4.8** Consider the code in the example below. When a Dog object is created its *__init__* method implicitly gets called. We never explicitly call the constructor on a Python object. Where does *__init__* get called in the JCoCo virtual machine?

```
mydog = Dog("Mesa")
```

*You can check your answer(s) in Section 4.34.8.*

```
1  public PyObject execute() {
2    this.PC = 0;
3    boolean handled = false;
4    JCoCo.pushFrame(this);
5    while (true) {
6      try {
7        inst = this.code.getInstructions().get(this.PC);
8        this.PC++;
9        opcode = inst.getOpCode();
10       operand = inst.getOperand();
11       switch (opcode) {
12         // instructions omitted
13         case SETUP_EXCEPT:
14           this.blockStack.push(-1 * operand);
15           opStack.push(new PyMarker());
16           break;
17       }
18     } catch (PyException ex) {
19       int exitAddress;
20       boolean found = false;
21       while (!found && !this.blockStack.isEmpty()) {
22         exitAddress = this.blockStack.pop();
23         if (exitAddress < 0) {
24           found = true;
25           if (!opStack.isEmpty()) {
26             PyObject obj = opStack.pop();
27             while (!obj.str().equals("Marker") && !opStack.isEmpty()) {
28               obj = opStack.pop();
29             }
30           }
31           this.opStack.push(ex.getTraceBack()); //The traceback at TOS2
32           this.opStack.push(ex); //The exception at TOS1
33           this.opStack.push(ex); //the exception at TOS
34           this.PC = -1 * exitAddress;
35           this.blockStack.push(0);
36         }
37       }
38       if (!found) {
39         ex.tracebackAppend(this);
40         throw ex;
41       }
42     } catch (Exception e) {
43       PyException ex =
44          new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
45          e.getMessage()+" while executing instruction "+inst.getOpCodeName());
46       ex.tracebackAppend(this);
47       throw ex;
48     }
49   }
50 }
```

**Fig. 4.41** A synopsis of exception handling in JCoCo

## 4.28 JCoCo Exceptions and Tracebacks

The *execute* method for *PyFrame* exits in one of two ways. Either the *RETURN_VALUE* instruction is executed, or an exception occurs that is not handled within this function. If an exception occurs, execution jumps to line 18 of the code in Fig. 4.41. All intentionally thrown exceptions thrown by JCoCo are *PyException* objects, so the catch block on line 18 will catch it.

JCoCo exceptions use the exception handling mechanism of Java to jump to the code starting on line 18 anytime a PyException is *thrown* in a JCoCo program, which would be any exception intentionally thrown by a CASM program. Upon entering the catch block on line 19 the code looks for an exception handler that may have been put in place to handle the exception in this PyFrame object. There may be an exception handler and there might not. JCoCo includes a block stack used for iteration and exception handling. The block stack records the exit points of loops and the addresses of exception handlers. For loops, the exit point is pushed on the block stack in case a *BREAK_LOOP* instruction is executed to break out of a loop. When an exception handler is put in place the location of the handler is indicated by a negative value on the block stack to differentiate it from loop exit points. Lines 13–16 show how an exception handler is set up within a frame. The *SETUP_EXCEPT* instruction pushes the exception handler address onto the block stack.

When an exception occurs the block stack is popped until the address of an exception handling block is found (i.e. a negative value is popped). The address of the exception handler is the negation of this negative value.

When an exception occurs it can occur anywhere within the code. In particular there could be operands left on the operand stack because the exception occurred in the middle of some other work. For instance, an exception might occur while preparing for a function call and arguments may be left on the operand stack. The PyMarker class serves to help clean up the operand stack after an exception is caught. When an exception handler is installed with the *SETUP_EXCEPT* instruction, a PyMarker object is pushed onto the operand stack. If a PyMarker object is popped during normal instruction execution, it is just thrown away. If an exception occurs the exception handling code on line 25–29 pops arguments from the operand stack until it is emptied or until a PyMarker object is found, thus cleaning up the operand stack.

If an exception handler was found in the JCoCo function's code, the exception is pushed onto the stack to get ready to jump to the exception handler code. Lines 31–33 seem a little strange until you remember that JCoCo maintains compatibility with Python 3.2 and disassembled Python code expects there to be three operands pushed onto the stack when an exception handler begins executing. Line 34 causes execution to jump to the first instruction of the exception handler. The Python virtual machine also assumes there is another exception handling block pushed on the blockStack. This is not needed by JCoCo, but to maintain compatibility line 35 pushes an entry onto the block stack.

If no exception handler is found then line 38 adds the current frame to the exception's traceback. The traceback is a list of all the PyFrame objects that are popped until an exception handler is found. If no exception handler is found, control returns to the main function where the traceback is printed.

Non-JCoCo Java exceptions may also be thrown by JCoCo code. This can happen in one of two scenarios. JCoCo may have a bug and if so, an exception may be thrown due to some unforeseen circumstance. The other reason a standard Java exception may be thrown would be due to the programmer attempting an illegal operation, like an arithmetic operation causing integer overflow for instance. If either of these

scenarios occur, then lines 42–49 will handle those exceptions. When a non-JCoCo Java exception occurs a PyException is created, the current frame is added to its traceback and the PyException is thrown. If this PyException is not caught anywhere within the CASM program then control will return to the main function in JCoCo.java and the exception's traceback will be printed as the source of the error.

## 4.29 Magic Methods

The JCoCo implementation of Python's virtual machine makes use of inheritance to reuse code in the implementation and to create *is-a* relationships between the objects manipulated by JCoCo. That type hierarchy is provided in Fig. 4.36. All of the JCoCo datatypes implement the PyObject interface which sits at the root of this type hierarchy.

Through inheritance every descendent of PyObjectAdapter contains a dictionary called *dict* that maps method names to methods as shown in Fig. 4.18. In Python, when a method is called, the method name is looked up in this dictionary to locate the code that corresponds to the method name. JCoCo mirrors this implementation to emulate the dynamic run-time typing behavior of Python. The dynamic lookup of methods occurs in the *callMethod* method of the PyObjectAdapter class as described in the related Sect. 4.13.

Figure 4.42 contains four methods that are defined on every type of value in JCoCo. For instance, any object can be converted to a string and all objects have a type within the hierarchy that can be retrieved. Notice that all these methods have the same signature. Each function or method in JCoCo (and Python) takes a list of objects as arguments and returns an object. Every JCoCo object implements methods with this signature and only this signature. The *__str__* and *__type__* methods are called magic methods by Python developers because they get called automatically by certain operators in Python. For instance, converting a PyObject to a string calls the *__str__* magic method to get a string representation of the object. Calling *type* on an object in a Python program results in calling the *__type__* method to get an object's type.

Calling *repr* on an object in a Python program calls the *__repr__* method. The *repr* function returns a string representation of an object that, when evaluated, would yield a copy of that same object. The *hash* function may also be called on any object, but only hashable objects implement the *__hash__* magic method with something other than throwing an exception.

Methods are the operations that can be performed on objects within the JCoCo type hierarchy. Magic methods are methods that get called as a result of either a built-in function call or some operator overloading in Python. The *__str__*, *__type__*, *__repr__*, and *__hash__* magic methods are added to the dictionary of methods for all objects by the PyObjectAdapter constructor. Figure 4.42 shows the methods that are supported by the PyObjectAdapter class for all subclasses. But, subclasses of PyObjectAdapter may add to the map of supported operations. For instance, the

```
public PyObjectAdapter() {
    name = "PyObject()";
    type = PyType.PyTypeId.PyClassType;
    PyObjectAdapter self = this;
    this.dict.put("__str__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return new PyStr(self.str());
        }
    });
    this.dict.put("__hash__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
            "TypeError: unhashable type: '" + self.getType().str() + "'");
        }
    });
    this.dict.put("__repr__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return self.callMethod("__str__", args);
        }
    });
    this.dict.put("__iter__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
            "TypeError: '" + self.getType().str() + "' object is not iterable");
        }
    });
    this.dict.put("__type__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return (PyObject) self.getType();
        }
    });
}
```

**Fig. 4.42** PyObjectAdapter's constructor

PyInt object's constructor calls the *funs* method to add a whole host of additional supported methods to integer objects as shown in Fig. 4.43.

The method name, called just *name* in the code in Fig. 4.20, is searched for in the dictionary. If it is not found, an exception is thrown. Otherwise, *mbr* is made to point at the code of the method (i.e. a PyCallable object). Line 6 calls the *__call__* method for this member function or method on the current object, returning whatever is returned from the call to the caller. The use of the dictionary maps names (i.e. strings) provided to JCoCo, to the methods of objects, which are implemented as PyCallable objects, within JCoCo. If the object does not have a method defined, the *callMethod* code gracefully handles this by throwing an exception which will result in a traceback being printed of the offending *CALL_FUNCTION* instruction.

Within JCoCo, magic methods get called as the result of many instructions. For instance, the *__add__* magic method gets called on an object as the result of executing the *BINARY_ADD* instruction. The *COMPARE_OP* instruction calls several different magic methods depending on the comparison operand of the instruction. Precisely which magic method gets called for a given instruction is detailed in the documentation provided in Appendix A.

**Practice 4.9** How can an object or class override the default behavior of a magic method like the *__str__* method without changing the JCoCo virtual machine itself?
*You can check your answer(s) in Section* 4.34.9.

```
1  public static HashMap<String, PyCallable> funs() {
2      HashMap<String, PyCallable> funs = new HashMap<String, PyCallable>();
3      funs.put("__hash__", new PyCallableAdapter() {...});
4      funs.put("__add__", new PyCallableAdapter() {...});
5      funs.put("__sub__", new PyCallableAdapter() {...});
6      funs.put("__mul__", new PyCallableAdapter() {...});
7      funs.put("__pow__", new PyCallableAdapter() {...});
8      funs.put("__truediv__", new PyCallableAdapter() {...});
9      funs.put("__floordiv__", new PyCallableAdapter() {...});
10     funs.put("__mod__", new PyCallableAdapter() {...});
11     funs.put("__eq__", new PyCallableAdapter() {...});
12     funs.put("__ne__", new PyCallableAdapter() {...});
13     funs.put("__lt__", new PyCallableAdapter() {...});
14     funs.put("__le__", new PyCallableAdapter() {...});
15     funs.put("__gt__", new PyCallableAdapter() {...});
16     funs.put("__ge__", new PyCallableAdapter() {...});
17     funs.put("__float__", new PyCallableAdapter() {...});
18     funs.put("__int__", new PyCallableAdapter() {...});
19     funs.put("__bool__", new PyCallableAdapter() {...});
20     funs.put("__str__", new PyCallableAdapter() {...});
21     return funs;
22 }
```

**Fig. 4.43** PyInt's additional magic methods

```
1  def main():
2    d = {}
3    d["hello"] = "goodbye"
4    d["dog!"] = "cat!"
5    d["young"] = "old"
6    s = "hello young dog!"
7    t = s.split()
8    for x in t:
9      print(x)
10   for x in t:
11     print(d[x])
12   for x in d.keys():
13     print(x, d[x])
14   for y in d.values():
15     print(y)
16   for key in d:
17     print(key, d[key])
18   print(type(d))
19   print(type(type(d)))
20 main()
```

**Fig. 4.44** dicttest.py

## 4.30 Dictionaries

Python implements a type called *dict*, short for dictionary, which is a map from *keys* to *values*. Dictionary objects are implemented as a hash table with O(1) get and set methods. A dictionary is created by using the braces around an optional list of key/value pairs. Line 2 of Fig. 4.44 shows an empty dictionary being created. Items are put in the dictionary using subscript notation. The key is the subscript and the value is the assigned value at the key's location. Lines 3–5 provide an example of storing key/value pairs in a dictionary. Line 11 uses subscript notation to look for a value corresponding to a key.

Dictionaries differ from lists because the subscript can be almost any type of value. Dictionaries are not limited to integer subscripts like lists. There are three requirements of a dictionary key. The key must be *hashable*. Hashing refers to deriving an integer from a value, as close to unique as possible. Keys in a dictionary must support an equality test. There must be a way of determining if two keys are equal. Finally, keys should not be mutable. Python lists are not suitable for keys because they are mutable. In JCoCo strings, integers, floats, tuples, and funlists are not mutable and therefore are suitable as keys in a dictionary. Funlists are a JCoCo specific functional programming list that is not a part of standard Python, but is supplied with JCoCo. Floats are not usually used as keys due to their being approximations of real numbers and the chance for round-off error in calculations. In general floats should

```
package jcoco;

public class PyDict extends PyPrimitiveTypeAdapter {
    private HashMap<PyObject, PyObject> map = new HashMap<PyObject, PyObject>();
    public PyDict() {
        super();
        initMethods(funs());
    }
    public void setVal(PyObject key, PyObject val) {...}
    @Override
    public PyType getType() {
        return JCoCo.PyTypes.get(PyTypeId.PyDictType);
    }
    @Override
    public String str() {...}
    public static HashMap<String, PyCallable> funs() {
        HashMap<String, PyCallable> funs = new HashMap<String, PyCallable>();
        funs.put("__getitem__", new PyCallableAdapter() {...});
        funs.put("__setitem__", new PyCallableAdapter() {...});
        funs.put("__len__", new PyCallableAdapter() {...});
        funs.put("__iter__", new PyCallableAdapter() {...});
        funs.put("keys", new PyCallableAdapter() {...});
        funs.put("values", new PyCallableAdapter() {...});
        return funs;
    }
}
```

**Fig. 4.45** Outline of PyDict.java

not be compared for equality and therefore, while immutable, they are not usually appropriate as keys in a dictionary.

The *dict* datatype is not included in the JCoCo implementation available to you on Github. This section describes the steps to add dictionaries as a case study of extending the JCoCo virtual machine. You must have the Java development environment installed on your computer to complete the steps described here. When you have completed the steps in this section the disassembled code from Fig. 4.44 will execute on JCoCo producing output similar to that of Python.

### 4.30.1 Two New Classes

Two new classes are required to support dictionaries; the PyDict class and the PyDictIterarator class. The PyDict class resembles the PyList class in some ways. The PyDict class must be added to the Java source code in a file called PyDict.java. An excerpt of the PyDict.java header file is given in Fig. 4.45.

There are several methods to be implemented to support dictionaries. The *__getitem__* method is given a key in the args vector and returns the correponding value. The *__setitem__* method maps a key to a value. The key is at index 0 in *args* and the value is at *args[1]*. The *__len__* method returns the size of the map. All of these methods use the HashMap called *map* and the methods of a HashMap are used in the implementation of the PyDict class.

The *map* instance variable is central to the PyDict implementation. A HashMap is part of the utility library of Java. There is a subtle implementation detail with the use of HashMap here. Java provides built-in support for hashing items. The *Object* class of Java includes a method called *hashCode* that is used to get a hash value for any hashable Java value. But, in JCoCo we wish to call the *__hash__* magic method to give the JCoCo assembly language programmer control over how an object is hashed. The *PyObjectAdapter* class comes to the rescue here by defining the *hashCode* method to call the *__hash__* magic method.

```
@Override
public int hashCode() {
    ArrayList<PyObject> args = new ArrayList<PyObject>();
    PyInt val = (PyInt) this.callMethod("__hash__", args);
    return val.getVal();
}
```

This means that the JCoCo PyDict implementation can just use the HashMap as you would in any Java program and the *__hash__* magic method will automatically get called when needed.

Many of the classes provided with Java have hashing functions defined for them already. The PyStr class can use the string hashing function provided by the *hashCode* method of PyObject. That *hashCode* function can be called in the *__hash__* magic method of PyStr to hash the string. Every type of object that could be used for a key in a dictionary must implement the *__hash__* method.

The HashMap needs to determine if two keys are equal as part of the hash table implementation. The HashMap needs to know if the key it is looking up matches one that it finds in the hash table. Again, PyObjectAdapter comes to the rescue. The HashMap automatically calls the *equals* method of Java Object to determine if the two keys are equal or not. The PyObjectAdapter overrides the *equals* method to call the *__eq__* magic method. So, the HashMap automatically makes calls to both the hash and equals magic methods. This means that the equals magic method must be implemented in any class that will be used as a key in a dictionary. Of course, this is already done for the built-in types of JCoCo.

The other class to be written is a PyDictIterator class that implements iteration over the keys of the dictionary. The PyListIterator can be used as an example in how to write this iterator. Remember that to terminate the iteration, the *PYSTOPITERATION* exception must be thrown once the iterator is exhausted. Here is how iteration is achieved over a Java HashMap object.

```
import java.util.Iterator;
import java.util.HashMap;
Iterator<HashMap.Entry<PyObject, PyObject>> it;
it = map.entrySet().iterator();
while (it.hasNext()) {
  HashMap.Entry<PyObject, PyObject> pair = it.next();
  System.out.println(pair.getKey());
}
```

```
1 def main():
2   d = { "Kent":"Denise",
3     "Sophus":"Addie"}
4   print(d)
```

**Fig. 4.46** Initializing a dictionary

This code must be divided up into the PyDictIterator implementation in a manner similar to the PyListIterator code.

### 4.30.2 Two New Types

In addition to the new classes, two new types must also be defined. The main module, JCoCo.java, contains a function called *initTypes*. The *dict* and *dict _keyiterator* types should be added as two new types to this function. To do this, two new values for the PyTypeID enum in PyType.java must also be defined; the *PyDictType* and *PyDictKeyIteratorType* values. This is a relatively simple addition to the code, but must be tied together with the implementations of the PyDict and PyDictIterator classes. Once these types are created, don't forget to set the instance functions in the type objects.

### 4.30.3 Two New Instructions

Finally, after disassembling the code in Fig. 4.44 a new instruction appears, the *BUILD _MAP* instruction. This instruction creates an empty dictionary and pushes it onto the operand stack.

Disassembling the code in Fig. 4.46 yields one other instruction. The *STORE _MAP* instruction expects three operands on the stack. The TOS element is a *key*, the TOS1 element is a *value* and the TOS2 element is a dictionary. The STORE _MAP instruction stores the *key/value* pair in the dictionary and leaves the dictionary on top of the operand stack when it completes. These two new instructions are implemented in PyFrame.java in the *execute* method. You can look at other examples of instructions to see how these two instructions should be implemented.

## 4.31 Chapter Summary

This chapter covered object-oriented, imperative programming in Java with C++ covered to a lesser degree. Advanced techniques including inheritance, polymorphism,

interfaces, generics, autoboxing and unboxing, inner classes and a few other important topics were covered with examples coming from the JCoCo virtual machine implementation.

Java and C++ are statically typed languages as compared to Python, which is a dynamically typed language. Static typing requires more work by the programmer when writing code, but also provides some level of assurance that code is type-correct. Python program type errors are not found until run-time. The C++ and Java compilers catch most type errors at compile-time.

C++ and Java share a lot of syntax, but they are distinct languages in many ways. C++ is especially suited to low-level and real-time implementations where performance is critical and you need access to the underlying hardware. C++ gives you complete control of when dynamically allocated data is freed. But with that responsibility comes the age old problem of memory leaks. C++ programs are prone to memory leaks and the C++ CoCo implementation is full of them because it is difficult to determine exactly when space can be freed in the virtual machine without implementing some form of garbage collection. C++ has many great programming features like templates, a large standard library, and compiler support for many hardware platforms.

Java programs benefit from garbage collection which is built into the JVM. Java also provides a unified type hierarchy for classes with Object at the root of the tree. C++ has no built-in class hierarchy. Java has built into it several nice programming features like auto-boxing and unboxing, threading support including synchronization support for all Java objects, inner class support, and support for separating interfaces from implementation.

Both C++ and Java serve as good examples of statically typed, object-oriented, imperative programming languages. They each have their benefits and shortcomings, but for the CoCo and JCoCo virtual machines, Java is the better-suited language providing garbage collection and a more unified approach to handling exceptions within the virtual machine. In the next chapter we introduce another programming paradigm while studying another statically typed language.

## 4.32 Review Questions

1. What does static type checking mean? Does C++ have it? Does Python have it? Does Java have it?
2. What are the names and purposes of the two programs that make up the Java environment for executing programs?
3. What is the number one problem that C/C++ programs must deal with? Why is this not a problem for Java and Python programs?
4. What does the *make* tool do and how does it work for C++ programs?
5. Is there an equivalent to the *make* tool for Java programs?
6. How does the C++ compiler distinguish between macro processor directives and C/C++ statements?

7. What is a namespace in C++? What is comparable to a namespace in Java? In Python?
8. What is the default executable name for a compiled C++ program?
9. What is separate compilation and why is it important?
10. What is dynamic linking? Does it happen in C++ or in Java? Why is it important?
11. Which environment has garbage collection built in, C++ or Java?
12. What are the advantages of garbage collection?
13. Are there any drawbacks to garbage collection?
14. What is a destructor and when is it needed?
15. What do you have to write to get a polymorphic method in C++?
16. What is the purpose of polymorphism?
17. What is the purpose of inheritance?
18. How do interfaces and classes differ in Java? How are they similar? How are they different?
19. What is an adapter class? Why are they useful?
20. What is a callback and how are they usually implemented in Java?
21. What are generics? Why are they convenient?
22. What is a template? How do you declare a vector in C++?
23. What is auto-boxing and unboxing?
24. How is a function represented as a value in Java?
25. What is an anonymous class?
26. What is the *type(6)* in JCoCo and Python? How about the *type(type(6))*? How about the *type(type(type(6)))*? Why isn't it interesting to go any further?
27. The JCoCo scanner is based on a finite state machine. How is the finite state machine implemented? What are the major constructs used by a finite state machine?
28. Does the JCoCo parser run bottom-up or top-down?
29. In JCoCo how are a PyCode object and a PyFunction object related?
30. What is a traceback and why is it important?
31. What is the purpose of a PyMethod class?
32. Arriving at hash values for hashable objects in Java is trivial. Describe how JCoCo determines hash values for objects in the implementation of PyDict objects.

## 4.33 Exercises

1. Alter the finite state machine of PyScanner.java to allow strings to include the escape character. Any character following the backslash, or escape character, in a string should be allowed. This project can be implemented by altering the PyScanner.java class to allow the escape character to appear within a string. Hint: Two extra states may be needed to implement this code. Note that JCoCo will already allow pretty much any character, including tabs and newline characters, to be included in a string constant. The only characters that pose problems are

single and double quotes. The escape character should not be included in the constant string, only the character that follows the escape character.

2. Implement true division and floor division for floats in JCoCo. Write a test program to thoroughly test these new operations supported by floats. The test program and the source code are both required for the solution to this problem. You may use the disassembler to help generate your test program.
3. Alter the JCoCo grammar to allow each line of a function's code to be either a JCoCo instruction or a source code line. Any source code line should be preceded by a pound sign, a line number, and a colon followed by the text of the source code line. A source code line would reflect a line from a source language other than JCoCo which was compiled to the JCoCo assembly language. Then, when an uncaught exception occurs in the JCoCo program, the traceback should be printed along with the source code line that caused the exception. This is a challenging exercise and requires changes to the scanner, parser, internal storage of PyCode objects, and traceback handling.
4. Add a dictionary object type to JCoCo by following the description at the end of this chapter. This project requires significant programming and there are pieces in the last part of the chapter that are left out. However, the provided code samples along with other similar code in the JCoCo project provides enough details to be able to complete it. When done, the successful project will be able to run the disassembled code from Figs. 4.44 and 4.46. The output should appear to be identical to the output produced by running the Python programs. However, the order of keys may be different since dictionaries are implemented with an unordered_map datatype.
5. Empty type calls produce *empty* results in Python but not in JCoCo. For instance, when *int()* is called in Python, the object 0 is created. In JCoCo this produces an error. Use Python to determine what should happen for all the empty type calls that JCoCo supports. Then modify CoCo so it will behave in a similar fashion.
6. Add a *set* datatype to JCoCo. Lookup the *set* datatype in Python documentation. Include support in your *set* datatype to support contructing a set, union, intersection, mutating union, and mutating set difference along with set cardinality, membership, and addition of an element to the set. Write a Python test program and disassemble it. Then run your test program to test your set datatype.
7. Modify JCoCo to allow instructions like *LOAD_FAST x* in addition to *LOAD_FAST 0*. Currently, the LOAD_FAST and STORE_FAST instructions insist on an integer operand. If an identifier operand were provided then the identifier must exist in the sequence of *LOCALS*. If it does not, the parser should signal an error. Internally, the LOAD_FAST and STORE_FAST instructions should not change. The conversion from identifier to integer should happen in the parser. Convert the *LOAD_GLOBAL*, and *LOAD_ATTR* instructions to allow either an identifier or integer operand in the same manner. Do not try to modify the *LOAD_CONST* instruction since it would be impossible to distinguish between indices and values for constants.

   This project is not too hard to implement. Labels are already converted to offsets in the parser in the *BodyPart* method. That code has to be modified slightly to

handle identifiers for things other than labels. The identifiers for the load and store instructions can be converted to integer operands in the *FunDef* function.

8. Currently the assembler has three different load instructions including *LOAD_FAST*, *LOAD_GLOBAL*, and *LOAD_DEREF* that all use indices into different lists as operands. Define a new pseudo *LOAD* instruction that lets you specify an identifier for a value to load. For instance *LOAD x* would result in scanning the *LOCALS* list for *x*. If *x* were found in the first position of the locals list, then the *LOAD x* would be changed to a *LOAD_FAST 0* instruction. Otherwise, if *x* was not in the list of locals, then the *GLOBALS* would be scanned next and if *x* were found there a *LOAD_GLOBAL* instruction would replace the *LOAD* pseudo instruction. If *x* was not found in the globals, then the cellvars could be scanned and finally the freevars. Create a *STORE* pseudo instruction as well for the *STORE_FAST* and *STORE_DEREF* instructions.
   Do not try to implement the pseudo instructions for any of the other load or store instructions. For instance, it would be impossible to know whether a *LOAD* referred to a *LOAD_DEREF* or a *LOAD_CLOSURE* if you tried to include *LOAD_CLOSURE* in your pseudo instruction.

## 4.34 Solutions to Practice Problems

### 4.34.1 Solution to Practice Problem 4.1

The Java compiler insists that if the class is called *Test* then the file must be *Test.java*. This is necessary because when class *A* is compiled and uses the *Test* class, the Java compiler must find the *Test* class. There is nothing included or imported into class *A* to tell the compiler where to look. Instead, Java looks for a file called *Test.java* if class *Test* is used in class *A*.

### 4.34.2 Solution to Practice Problem 4.2

The C++ compiler uses macro processor directives to explicitly include header files which declare classes and other entities. The header file names are explicitly provided in the *include* macro processor directive. The C++ compiler does not need to infer the name of the module from the name of the class like Java does.

### 4.34.3 Solution to Practice Problem 4.3

In C++ programs the name of the executable for the program is passed in *argv[0]*. So the value of *argc* is always at least one. In a Java program the program consists of a collection of .class files. The main function must be defined inside the public

class for one of these .class files, so the name of the main module is always known by the programmer and therefore is not passed as an argument.

### 4.34.4 Solution to Practice Problem 4.4

Polymorphism occurs in Python because methods are looked up by name at run-time. This leads to run-time type checking, not compile-time as is supported by C++ and Java. C++ and Java are statically typed languages. Python is dynamically typed. Further, Python supports inheritance, but the purpose of inheritance in Python is only for code re-use. Polymorphism is not related to inheritance in Python programs since polymorphism occurs because of the late, just in time, lookup of methods in an object's attribute dictionary.

### 4.34.5 Solution to Practice Problem 4.5

The confusion may occur when an *ArrayList* of *Integer* was declared. When calling *a.remove(1)* will autoboxing occur? The answer is no, in this case because the *ArrayList* class contains a method with *int* as a parameter Java will choose the method with the closest argument types when there are multiple to choose from. Nevertheless, the programmer must be aware of this to correctly choose the right *remove* method. If the *Object* argument version is really the one to call, then it must called as *a.remove(new Integer(1))* to get the correct *remove* called. Boxing must be explicitly called in this case. No autoboxing will occur.

### 4.34.6 Solution to Practice Problem 4.6

It's the LOAD_BUILD_CLASS instruction. This instruction loads the built-in function onto the operand stack so it can be called to dynamically build a class.

### 4.34.7 Solution to Practice Problem 4.7

The bound variables are *f*, *x*, and *y*. They are bound because the name of the function is always defined and the parameter name is given the value of any argument passed to the function. The variable *y* is bound to the same value as *x* because *y* appears on the left side of an assignment statement.

The free variables are *aVal* and *lstInts*. These values have to be supplied in the environment by forming a closure before the function can be executed.

### 4.34.8 Solution to Practice Problem 4.8

The __init__ gets called during object instantiation on line 52 of Fig. 4.39.

### 4.34.9 Solution to Practice Problem 4.9

Since magic methods are looked up at run-time (i.e. dynamically) then at any time before the magic method, the object can have its magic method implementation replaced by an alternative implementation. There is no modification necessary to the JCoCo Java code.