laptop, tablet, or smartphone.

PUTTING THE CLOUD TO WORK

In Chapter 10, I created a simple application that searches a dictionary for certain words. I created a single core application and then one that leverages multiple cores. As you can see in Figure 11.2, I have executed the two programs that perform the simple dictionary search program. I executed them from my iPad with the application running in the cloud. This code is unchanged from the version created in Chapter 10:

sp.py is the single processing version
mp.py is the multiprocessing version

In Figure 11.3, I execute both the Simple Single Core and Multi-Core application from my desktop browser. As you can see the Multi-Core runs a bit faster even though I am only running the multiprocessing application on two-core cloud computers at Python Anywhere.

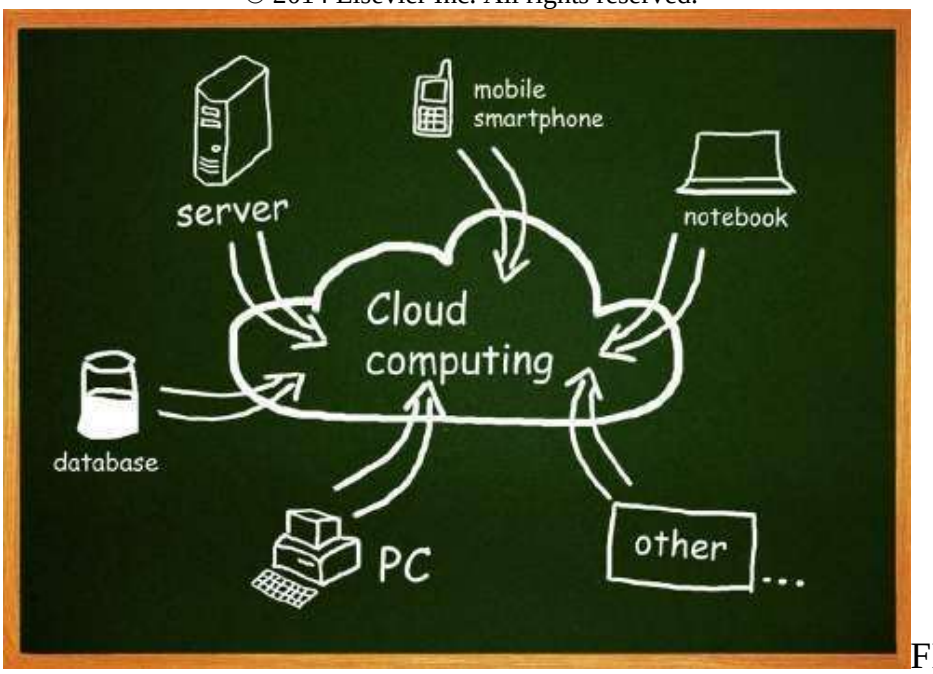


FIGURE 11.1
Typical cloud configuration.

Note that in these examples I have not changed a line of code to execute the Python code in the cloud. Since the code is executed by a standard Python

interpreter and I have used Python Standard Libraries the code just runs. If you utilize third-party libraries then you will have to get those added to the cloud-based Python installations (this is not impossible to do), but it is much easier if we stick with the Standard Library. I am using the cloud service Python Anywhere or www.pythonanwhere.com as shown in Figure 11.4. Python Anywhere is a terrific place to start experimenting with Python applications in the cloud. Signup is free for a minimal account, giving you access to a couple of cores and storage for your programs. I have a $12/month plan and this gives me 500 GB of storage and 20,000 CPU seconds allowed per day. The seconds are seconds of CPU time—so any time that any processor spends working on your code, internally it is measured in nanoseconds. Most online cloud services reference the Amazon definition of the CPU second which is: “the equivalent CPU capacity of a 1.0–1.2 GHz 2007 Opteron or 2007 Xeon 64 bit processors.”

CLOUD OPTIONS

Many cloud options exist that allow access to as few as 2 and as many as 1000 cores on which to execute. Table 11.1 provides an overview of just a few that you could investigate further (note, some are ready to run Python code, others require you to set up your own Python environment). In addition, some have a custom application interface or application interface (API) that you must use for multiprocessing, while others use a native approach.

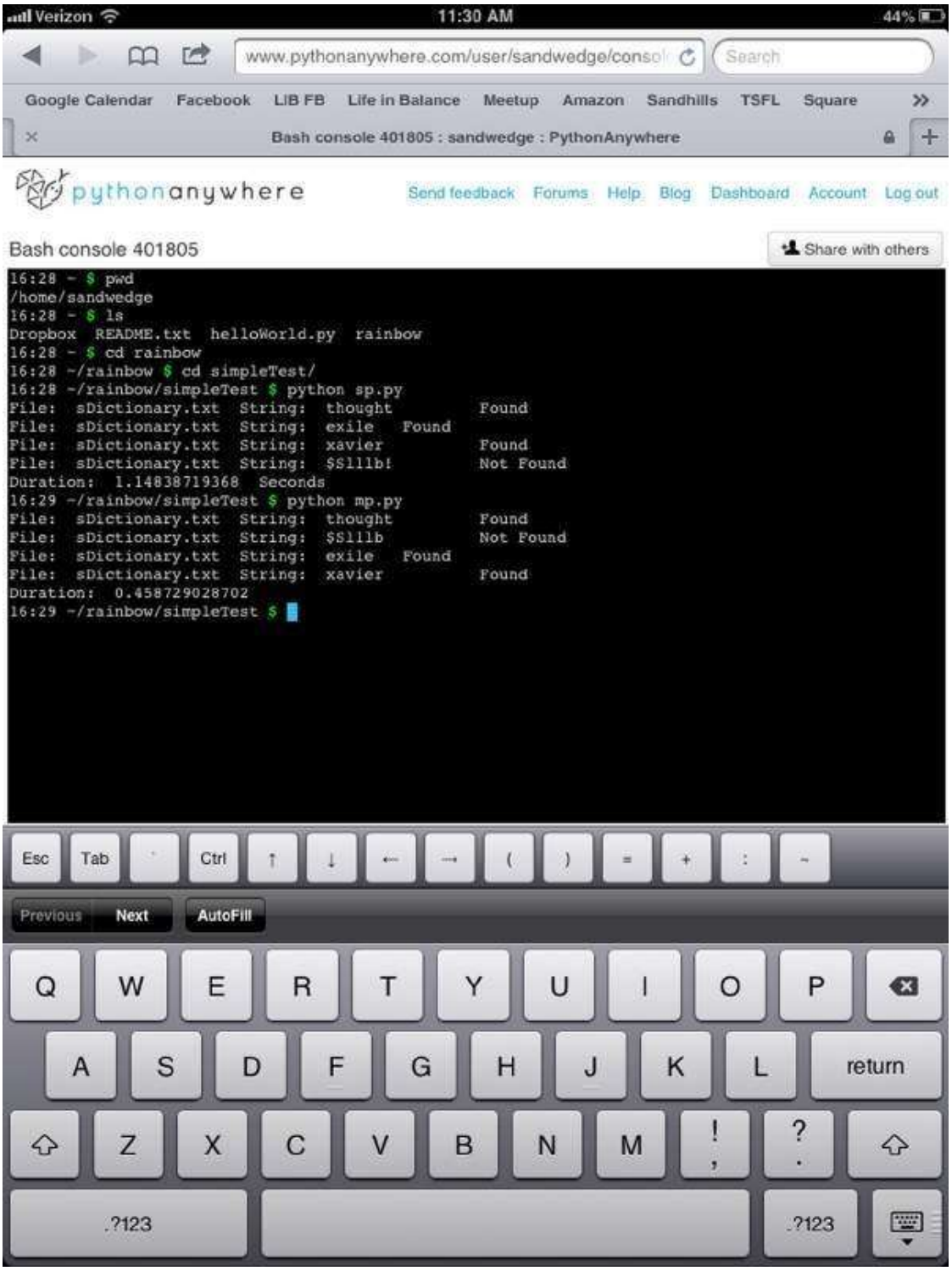


FIGURE 11.2
Cloud execution from iPad.

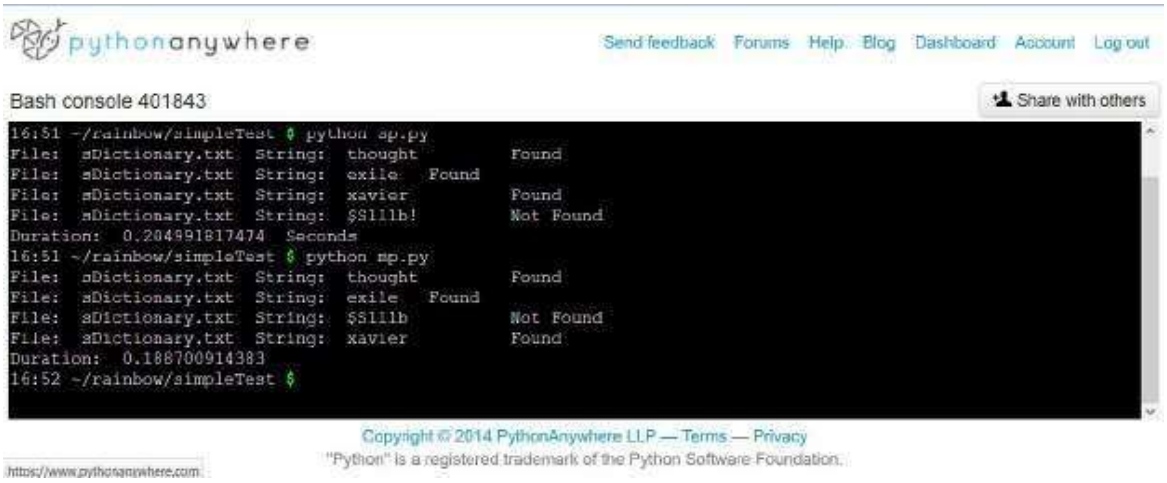


FIGURE 11.3
Desktop execution of the simple and multiprocessing Python applications executing in the cloud.

FIGURE 11.4
Python Anywhere Home Page.

Table 11.1 A Few Python Cloud Options Cloud Service URL Notes
[PythonAnywhere] pythonanywhere.com
[PiCloud] picloud.com
[Digital Ocean] digitalocean.com
Others Amazon, Google, ATT,

IBM, Rackspace just to mention a few
Runs native Python code for version 2.6, 2.7, 3.3
Figure 11.5
Runs native Python code, but requires the import of their cloud module for

multiprocessing.
See Figures 11.6 and 11.7
Requires you to install a Python package for the environment and your application. See Figures 11.8 and 11.9
As you expand your applications these services offer a variety of solutions

CREATING RAINBOWS IN THE CLOUD

There are considerable trade-offs and options available for creating high performance Python-based investigation platforms. One of the interesting applications that is suited nicely for cloud execution is the generation of Rainbow Tables discussed and experimented with in Chapter 10. Moving both the single and multiprocessing versions of this application to the cloud requires some considerations. First, I have decided to use Python Anywhere to demonstrate this capability. As I mentioned earlier, it is a great way to get started in the cloud, because the environment executes native Python applications for version 2.6, 2.7, and 3.3. Since I have made sure to only use Standard Library modules and core language elements, portability to the

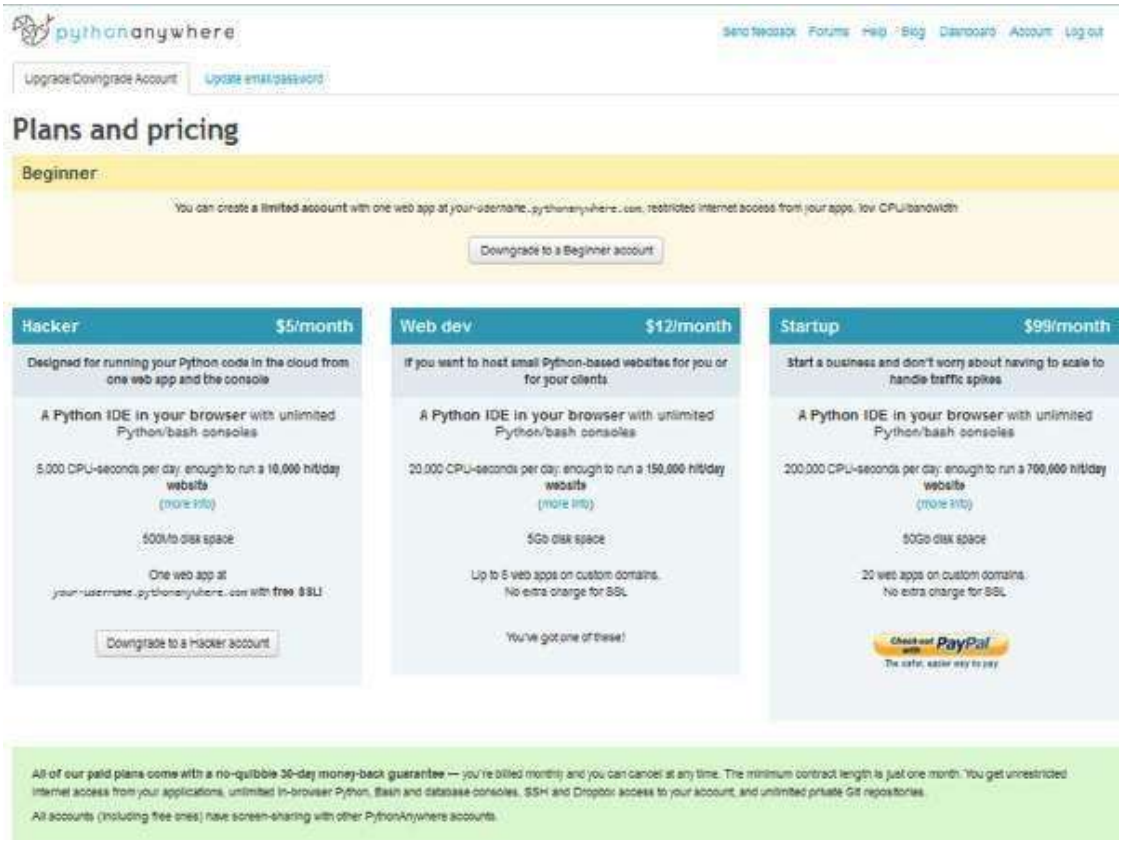


FIGURE 11.5

Python Anywhere Plans.

FIGURE 11.6
PICloud Home Page.

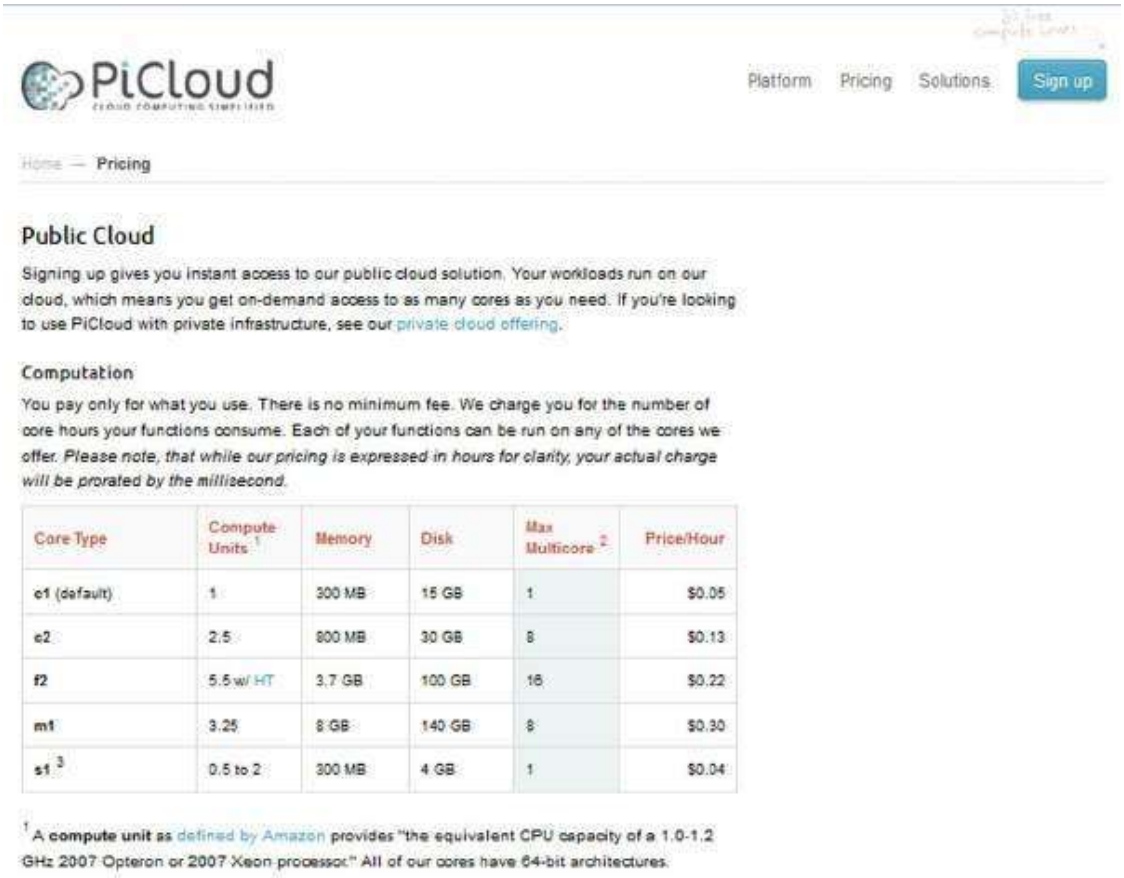


FIGURE 11.7
PICloud Plans.

FIGURE 11.8
Digital Ocean Home Page.

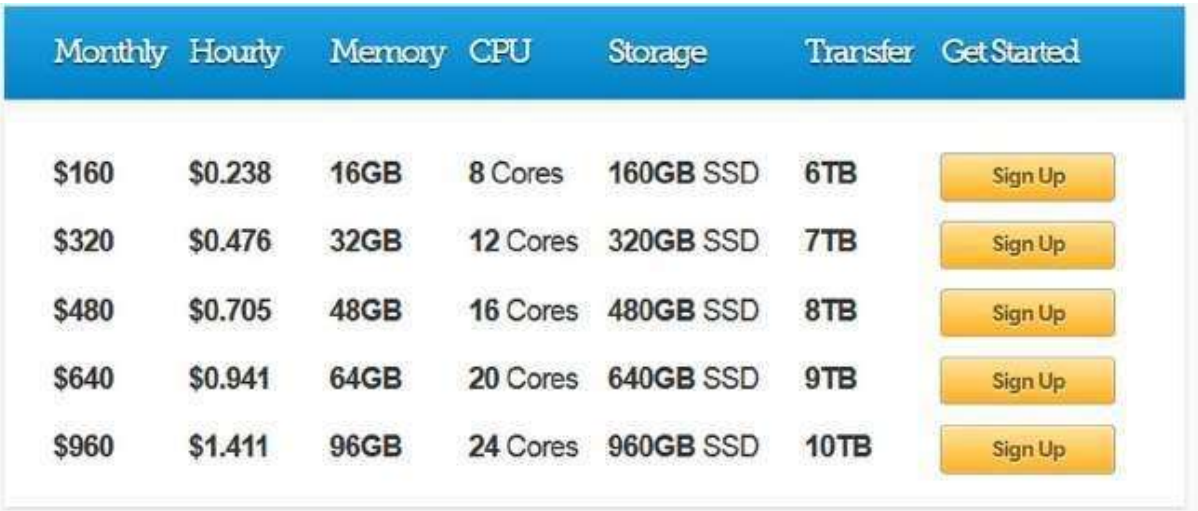


FIGURE 11.9
Digital Ocean Plans.

Python Anywhere cloud is pretty simple. However, I am going to make a couple of significant changes to the experimental code that I developed in Chapter 10:

(1) Minimize the memory usage within the programs by eliminating the use of the lists and dictionaries.
(2) Simplify the characters used to keep the resulting password generation reasonable
(3) Expand the generated password lengths to 4- to 8-character passwords

The resulting code listings for both the single and multiprocessing versions are shown here:
Single Core Rainbow

```
# Single Core Password Table Generator
# import standard libraries

import hashlib # Hashing the results
import time # Timing the operation import sys
import os
import itertools # Creating controled combinations

#
# Create a list of characters to include in the # password generation
#

chars ¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

```
# Define a hypothetical SALT value
SALT¼"&45Bvx9"
# Define the allowable range of password length PW_LOW¼4
PW_HIGH¼8

print'Processing Single Core'
print os.getcwd()
print'Password Character Set:', chars
print'Password Lenghts:', str(PW_LOW),'-', str(PW_HIGH)

# Mark the start time startTime¼time.time()
# Open a File for writing the results

try: # Open the output file
fp¼open('PW-ALL','w')

except:
print'File Processing Error' sys.exit(0)

# create a loop to include all passwords # within the allowable range
pwCount¼0
for r in range(PW_LOW, PW_HIGH+ 1):
#Apply the standard library interator
for s in itertools.product(chars, repeat¼r):
# Hash each new password as they are # generated

pw ¼".join(s) try:
md5Hash¼hashlib.md5()
md5Hash.update(SALT +pw)
md5Digest¼md5Hash.hexdigest()

# Write the hash, password pair to the file fp.write(md5Digest +"+pw +'\n')
pwCount+¼ 1
del md5Hash

except:
print'File Processing Error'
# Close the output file when complete fp.close()
```

When complete calculate the elapsed time elapsedTime¼time.time() - startTime print'Single Core Rainbow Complete' print'Elapsed Time:', elapsedTime print'Passwords Generated:', pwCount print

Multi-Core Rainbow
Multi-Core Password Table Generator
import standard libraries

import hashlib # Hashing the results import time
import os
import itertools
import multiprocessing # Timing the operation

Creating controled combinations # Multiprocessing Library

Create a list of characters to include in the # password generation
#

chars¼ ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
Define a hypothetical SALT value SALT¼"&45Bvx9"
Define the allowable range of password length
PW_LOW¼4 PW_HIGH¼8
def pwGenerator(size):

pwCount ¼0
create a loop to include all passwords # within range specified

try:
# Open a File for writing the results fp¼open('PW-'+str(size),'w')
for r in range(size, size+ 1):
#Apply the standard library interator

for s in itertools.product(chars, repeat¼r): # Process each password as they are # generated
pw¼''.join(s)

# Perform hashing of the password md5Hash¼hashlib.md5()
md5Hash.update(SALT +pw)
md5Digest¼md5Hash.hexdigest()

```
# Write the hash, password pair to the file fp.write(md5Digest +"+pw +'\n')
pwCount+¼ 1
del md5Hash

except:
print'File/Hash Processing Error'
finally:
fp.close()
print str(size),'Passwords Processed¼', pwCount

#
# Create Main Function #

if __name__ ¼¼'__main__':

print'Processing Multi-Core'
print os.getcwd()
print'Password string:', chars
print'Password Lengths:', str(PW_LOW),'-', str(PW_HIGH)

# Mark the starting time of the main loop startTime¼time.time()
#create a process Pool with 5 processes
corePool¼multiprocessing.Pool(processes¼5)
#map corePool to the Pool processes
results¼corePool.map(pwGenerator, (4, 5, 6, 7, 8))
elapsedTime¼time.time() - startTime
# When complete calculate the elapsed time

elapsedTime¼time.time() - startTime print'Multi-Core Rainbow Complete'
print'Elapsed Time:', elapsedTime print'Passwords Generated:', pwCount print
```

You can see the results of executing both the single core and multi-core solutions on the Python Anywhere Cloud, respectively, in Figures 11.10 and 11.11. I have also included the execution from my Linux box in Figure 11.12. The actual performance will vary of course based on many factors. Since my Linux box is dedicated and is running a quad-core processor at 3.0 GHz it outperforms the cloud service. The single core and multi-core results demonstrate a proportional result with the multi-core solution outperforming the single core solution as expected (see Table 11.2).

FIGURE 11.10
Python Anywhere Single Core Execution Results.

FIGURE 11.11
Python Anywhere Multi-Core Execution Results.

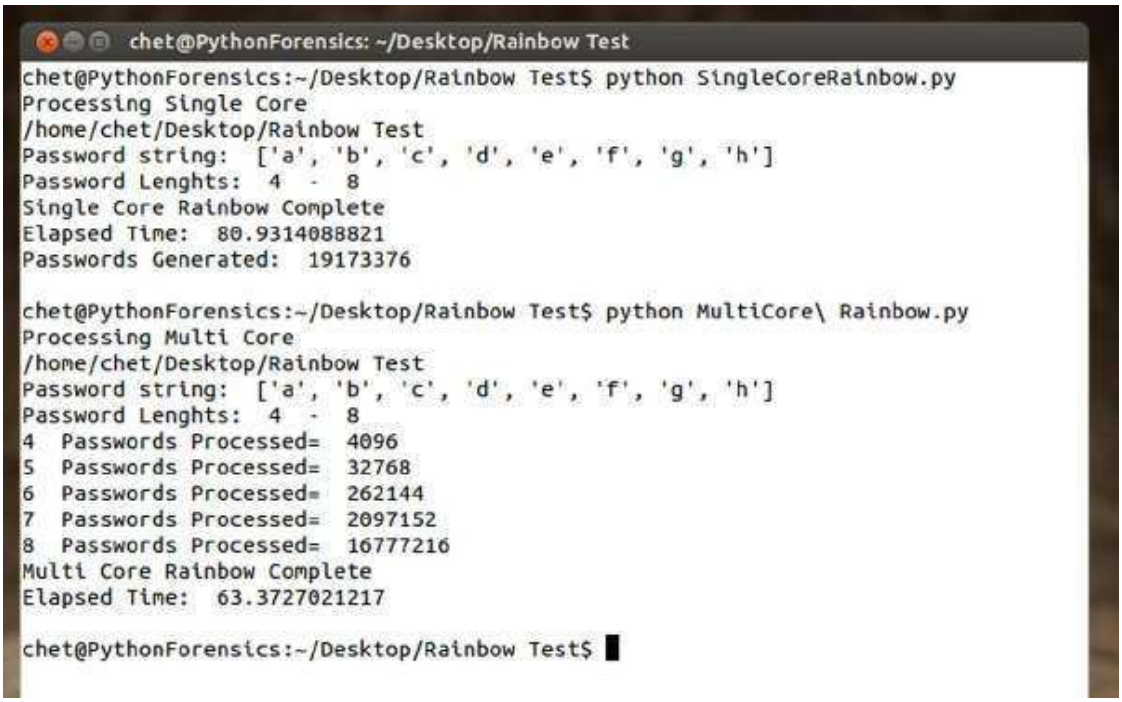


FIGURE 11.12
Standalone Linux Single/Multi-Core Execution Results.
Table 11.2 Summary of Execution Results Execution
configuration Passwords generated Time to and processed process(s) Passwords per second

Standalone Quad Core Linux
Single Core
Multi-Core

Python Anywhere Single Core
Multi-Core
19,173,376 80.93 19,173,376 63.37 236,913 302,562

19,173,376 210.99 19,173,376 142.93 90,873 134,145

PASSWORD GENERATION CALCULATIONS

One question you might be asking at this point is how many unique combinations of passwords are there? In order to be reasonable, let us start with the number of possible 8-character passwords by just using lowercase letters. The answer is shown in Figure 11.13 calculated by elPassword [elPassword]. In

Figure 11.14, elPassword calculates the number of unique 8-character passwords using upper and lowercase letters, numbers, and special characters.

FIGURE 11.13
elPassword 8-character combinations of lowercase letters.
Password Generation Calculations 301

FIGURE 11.14
elPassword 8-character full ASCII character set.

Using the online resource from LastBit, [LastBit] along with our best performance of 302,000 passwords per second, we can calculate the length of time required for a brute force attack. In Figure 11.15–11.18, I performed four separate runs. The first two using all lowercase characters with 1 and 100 computers, and the last two using the full ASCII set with 100 and 10,000 computers, respectively. Try them out for yourself.

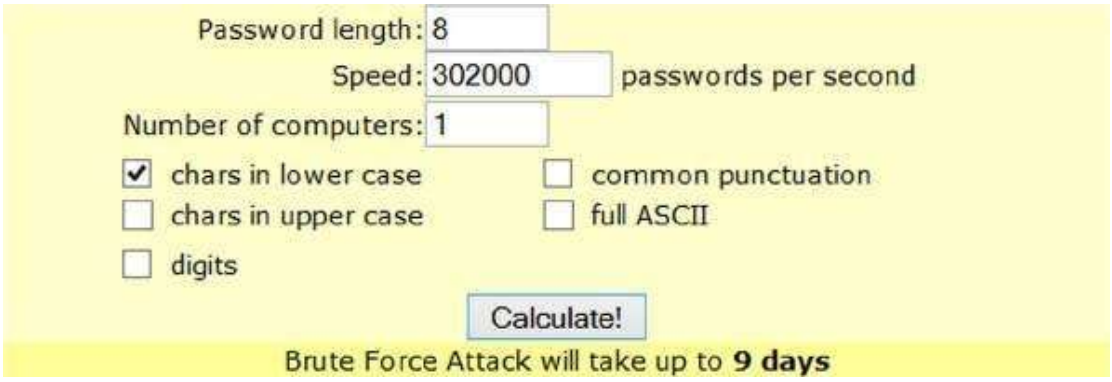


FIGURE 11.15
Last Bit calculation lowercase using 1 computer.

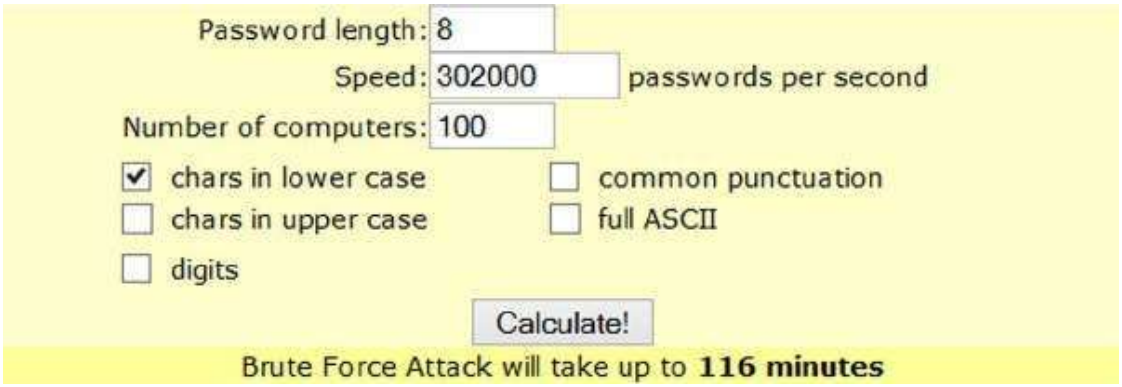


FIGURE 11.16
Last Bit calculation lowercase using 100 computers.

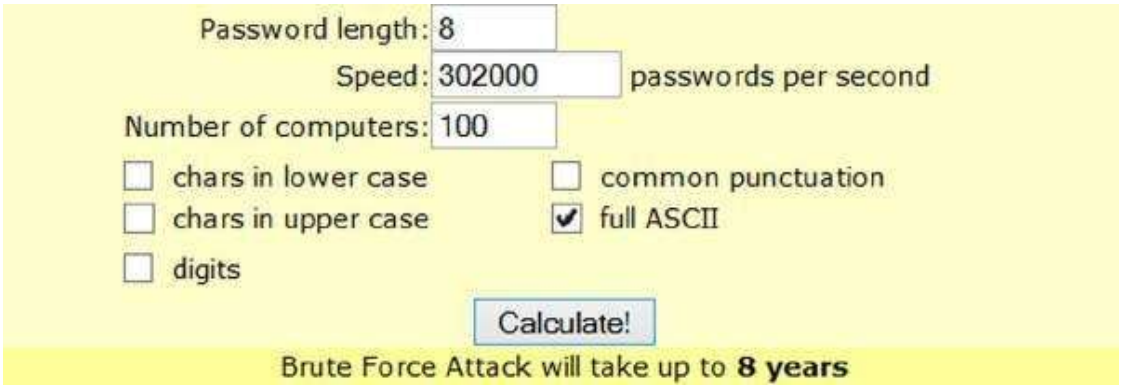


FIGURE 11.17
Last Bit calculation ASCII set using 100 computers.

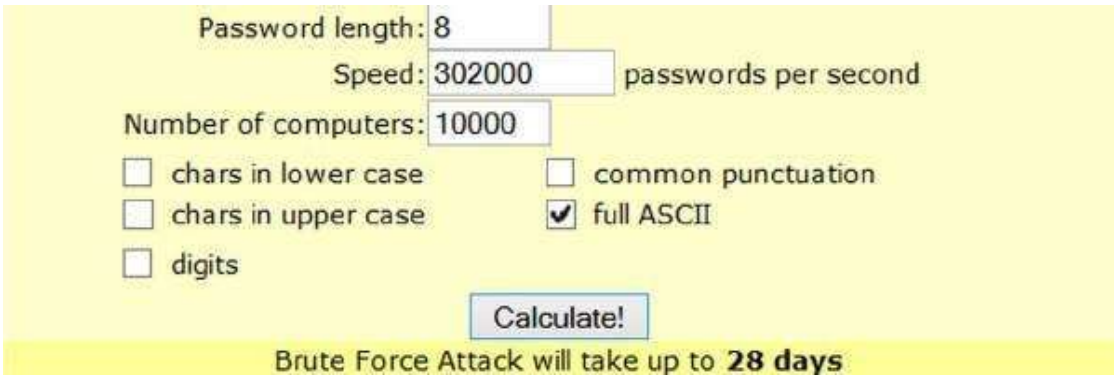


FIGURE 11.18
Last Bit calculation ASCII set using 10,000 computers.

CHAPTER REVIEW

In this chapter I introduced Python Anywhere, a cloud service that runs native Python code in the cloud. I demonstrated how simple it is to run native Python code in the cloud from multiple platforms. I then modified the Rainbow Table password generator to minimize memory usage, reduced the characters, and expanded the generation

to include password lengths of 4- to 8-character solutions. I then examined the performance of each result to determine how long it would take both on a higher performance Linux platform and in the cloud. I took those results and extrapolated them in order to determine the time and computers needed to reasonably crack 8-character passwords.

SUMMARY QUESTION/CHALLENGE

1. What other applications would be useful to execute within cloud environments that would benefit the forensic and investigation community?
2. As of this writing, Intel and AMD are experimenting with single processor core counts of 16, 32, 64, and 96 cores. If we extrapolate just a few years in the future it would not be unreasonable to expect the achievement of 1000-core CPUs. How will this change our ability to generate and hash passwords, crack encryption, or search data?
3. Develop and test your own multi-core solution on your desktop environment. Design them such that they can be easily deployed to a cloud platform like Python Anywhere. Establish a free account on Python Anywhere and experiment with your single and multi-core solutions.

Additional Resources

http://www.pythonanywhere.com . http://www.picloud.com.
http://www.digitalocean.com.
http://projects.lambry.com/elpassword/. http://lastbit.com/pswcalc.asp.

CHAPTER

Looking Ahead 12

CHAPTER CONTENTS

INTRODUCTION

The world of digital investigation and computer forensics is approaching 25 years in age. I had the privilege of working with Ron Stevens from the New York State Police who was one of the early pioneers and one of the first troopers in the country to successfully collect evidence from computers and use that evidence to convict those involved in criminal activity. “As one of the first law enforcement agencies to respond to the threats posed by the techno-criminal, the New York State Police launched the Computer Crime Unit in 1992” (Stevens, 2001).

I also had the privilege of serving as the primary investigator on the first digital forensic research effort sponsored by the Air Force Research Laboratory in 1998. The effort was conceived by two pioneers in the field working at the U.S. Air Force Research Lab, Joe Giordano and John Feldman. The effort was entitled: Forensic Information Warfare Requirements and resulted in a comprehensive report that outlined a set of requirements for the field moving forward. (FIW Report, 1999)

However, it would be a mistake to think of digital investigation, cybercrime investigation, or computer forensics as a mature science or discipline today. The following is an excerpt from a presentation I gave at the First Digital Forensic Research Workshop in 2001 (DFRWS, 2001).

Digital Evidence Fundamentals

– Digital evidence is vast, complicated, and can be easily manipulated, concealed, destroyed, or never found
– Connecting the dots can be an arduous process, fraught with uncertainty and in many cases is inconclusive
– The distributive nature of cyber-crime and cyber-terrorism makes tracing the perpetrators, the victims, and the technology used to execute the attack or crime difficult
– Our results will likely be challenged, thrown out, ignored, misunderstood, and constantly questioned

Cyber-forensic technologies must be:

– Reliable
– Accurate
– Nonreputable
– Easy to use
– Foolproof
– Secure
– Flexible
– Heterogeneous
– Distributed
– Automatic
– Free, or at least cheap

Fundamental questions—Cyber-forensic technologies

– Who collected the digital evidence?
– With what tool or technology?
– By what standard or practice is this based?
– Who audits and validates the practices?
– How are the identities of the digital detectives bound to the digital evidence?

– How was the evidence handled once identified?
– How is the evidence validated, and by whom?
– For how long is it valid?
– How is it stored and secured?
– How is the integrity of digital evidence assured?
– What technology was used to assure it?
– Why do I trust the tool or technology?

– Who developed it?
– Where and under what conditions was it developed?
– What underlying software and hardware does the technology rely on?
– Who validated or accredited the technology and the process?
– Which version(s) are accredited?
– Who trained and accredited the users

– Is the evidence distinctive?
– Is the evidence privileged?
– Is the evidence corroborated?

– When was the file created, modified, or destroyed?
– When was the transaction executed?
– When was a message transmitted or received?
– When was the virus or worm launched?
– When was the cyber-attack initiated?
– How long after the reconnaissance stage was completed did the attack commence?
– In what time-zone?
– At what point was the system log still valid?
– Did the suspect have the opportunity to commit the crime?

Fundamental questions

– Where is the suspect in cyber-space?
– How can I trace his or her steps?
– Technically
– Legally
– Where are they likely to strike next?
– Are they working with accomplices or insiders?
– What capabilities do they posses?

– Bandwidth
– Computing power
– Savvy
– Resources
– Have we seen them before?
– Are they more sophisticated now vs. a year ago?
– Who are their accomplices?

Summary

– We must focus our attention on identifying and addressing the key fundamental elements of digital forensics and digital evidence
– We must work together to build technologies that address the issues surrounding these fundamental elements
– We need to perform research where the goal is to use the science of forensics to help answer the harder questions surrounding cyber-criminals and terrorists
– Their location, sophistication, likely next targets
– What do we need to do to stop them?
– How do we improve our defenses against them?

The scary part about this short trip down memory lane is that I could give this presentation today, and it would still be for the most part relevant. The big question is where we go from here and how will Python Forensics play a role?

WHERE DO WE GO FROM HERE?

This book has identified key areas where research and development of new solutions are possible when combining core challenge problems with the Python language. It could certainly be argued that other languages could produce viable solutions to these challenges. However, the questions are:

– Would these alternative solutions be open source and free?
– Would they be cross platform ready (Windows, Mac, Linux, Mobile, and in the Cloud)?
– Would they be accessible and readable by anyone?
– Would they have the global worldwide support?
– Do they support a collaborative environment that would provide an on-ramp for computer scientists, social scientists, law enforcement professionals, or students new to the field?

The next big steps from my view are as follows:

(1) Creation of a true collaborative environment where people can share information (challenge problems, ideas, and solutions).
(2) Obtain nonintimidating support as technology is advanced in support of new investigative challenges.
(3) Development of a repository of programs and scripts that could be downloaded, applied to real word problems, expanded, and improved.
(4) Integration with on-demand training courses to dive deeply into core areas of Python and forensics.
(5) A validation/certification process that would allow third-party organizations (such as NIST) to validate Python supplied solutions for use by law enforcement. Once validated they could be safely utilized on real cases. Once the process has been created much of the validation work could be automated through the use of standardized forensic test images to accelerate the validation process.
(6) Vendors could supply application interfaces that open the possibility to integrate new Python-based solutions into existing forensic technologies. This would actually improve the capabilities of vendor solutions, allow them to address new issues in a more timely fashion, and make their products more valuable in the marketplace.
(7) The creation of a cloud-based experimental platform that has thousands (or even hundreds of thousands) of processor cores, petabytes of storage, and terabytes of memory that could be applied and harnessed to solve computationally difficult problems. In addition, this environment would be open to academia and students in order to rapidly advance classroom problems. Collaboration across universities, colleges, practitioners, vendors, and researchers to solve truly hard problems. The environment could provide a competition surface for new innovations, and teams could compete on the national and international stage. Benchmarks for specific solution types could be created in order to understand the performance characteristics of various solutions.
(8) Key challenge problems for consideration:
Throughout this book, I provided Python-based examples for several key areas. These examples provide the basics, however, much more work is needed. I have identified some of the key challenges that require additional work and focus.
a. Advanced searching and indexing: search and indexing are certainly core elements during any investigation. However, improving the speed, accuracy, and relevance of search and index results is necessary. Investigators need solutions

that can provide the following in a timely fashion: i. Search and index results that deliver information that is relevant to their case. This rich search/index capability must uncover information that is not obvious or is missed by present-day technologies. For example, search results that include time and spatial connections into clear view.
ii. Search and index results that connect information from multiple cases together, identifying connections between accomplices, Internet, phone, time, location, and behavioral analysis that were previously unconnected.
b. Metadata extraction: Images and multimedia content contain a plethora of metadata, including but not limited to time, date, the device they were created on, location information, subject content, and much more. The extraction, connection, and reasoning about this information is today left to the investigator. New innovations bring the promise of rapidly extracting and connecting this information to provide a broader more comprehensive view of the crime scene.
c. Event synchronization: According to Statistic Brain (2014) in 2013 there were an average of 58 million tweets per day from over 645 million registered Twitter users. In addition, (Statistic Brain, 2014) 5.9 Billion Google searches per day and over 2 trillion searches for the year 2013 were recorded. This is only a fraction of the total Internet events that occur each day and each year. Our ability to synchronize and reason about events whether from the Internet, a corporate network or even an individual desktop may seem beyond comprehension. We need to develop new innovations that can make sense, isolate, and provide conclusive proof about such actions and events.
d. Natural language: The Internet has certainly broken down boundaries and provided interaction and communication instantly across the globe. As I just scratched the surface of Natural Language Processing (NLP) in Chapter 7, you can see the potential power of being able to process language. The application for NLP for the extraction of meaning, determining authorship, deciphering intent are all within our grasp. Expanding these technologies to process a wide range of languages, improving deductive reasoning, extracting persons, places and things, and assessing likely past, present or future actions are possible.
e. Advancement in Python. The printed examples and source code provided in this book were developed for Python 2.7.x in order to ensure the broadest compatibility across computing platforms. However, all the examples will also be available online for download with solutions for 2.7.x and 3.3.x. Python and third-party Python libraries continue to expand at the speed of the Internet. Some even claim Python will be the last programming language you will ever have to learn. I think this is a bit overstated however, some of the attributes of the language will be fundamental to future languages. At the time of this writing

February 9, 2014 to be exact, Python Version 3.4 was released. According to the Python Programming Language Official Web (www.python.org), the major language improvements included: 1. a “pathlib” module providing object-oriented file system paths 2. a standardized “enum” module
3. a build enhancement that will help generate introspection information for built-ins
4. improved semantics for object finalization
5. adding single-dispatch generic functions to the standard library
6. a new C API for implementing custom memory allocators
7. changing file descriptors to not be inherited by default in subprocesses
8. a new “statistics” module
9. standardizing module metadata for Python’s module import system 10. a bundled installer for the pip package manager
11. a new “tracemalloc” module for tracing Python memory allocations 12. a new hash algorithm for Python strings and binary data 13. a new and improved protocol for pickled objects
14. a new “asyncio” module, a new framework for asynchronous I/O As you can see the language evolution continues at a brisk pace along with the development of new and improved third-party Python libraries that help accelerate forensic and digital investigation tools. There are so many third-party libraries and tools, I decided to list just my top 10.
1. Pillow. A new library that builds off of the more traditional Python Image Library for those that process and examine digital images.
2. wxPython. For those of you that need to build cross platform graphical user interfaces this is my preferred toolkit.
3. Requests. One of the best http interface libraries.
4. Scrapy. If your forensics investigations require you to perform web scraping, this library will help you build new and innovative approaches.
5. Twisted. For those needing to develop asynchronous network applications.
6. Scapy. For those who perform packet sniffing and analysis, scapy provides a host of features that can speed your development.
7. NLTK. Natural Language Toolkit—This toolkit is vast and for those investigating text and language constructs, it is a must.
8. IPython. When experimenting with new language elements, libraries, or modules this advanced Python Shell will assist you in every aspect of your work or struggles.
9. WingIDE. This is not a library, but this Integrated Development Environment gets my vote for the best IDE. The Professional version provides even the most savvy researcher with the tools they need. 10. Googlemaps. Many forensic

applications collect geographically tagged information, this package allows you to easily integrate with the Google mapping system.
f. Multiprocessing: In order to effectively attack any of these areas, our ability to access the power of the latest processors and cloud-based solutions is essential (Figure12.1). “The International Technology Roadmap for Semiconductors 2011,” a roadmap forecasting semiconductor development drawn up by experts from semiconductor companies worldwide, forecasts that by 2015 there will be an electronics product with nearly 450 processing cores, rising to nearly 1500 cores by 2020 (Heath).
What is available today from the two major processor manufactures, Intel® and AMD® are depicted in Figures 12.2 and 12.3. For under $1000 you can own these processors and for under $3000 you can build a system with two processors yielding 20-32 cores, 64 GB of memory and a couple of terabytes of storage. This is certainly a step toward multicore and multiprocessing solutions, and when coupled with well-designed multicore

FIGURE 12.1
Multiprocessing in the Cloud.

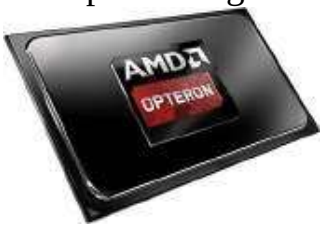


FIGURE 12.2
AMD 6300 Series 16 Core Processor.

FIGURE 12.3
Intel Xeon E7 Series 10 Core 20 Thread Processor.

Python applications could take a giant step toward advancing the state-of-theart. Once we see these production solutions produce 64, 128, 256, and 1024 cores on a single processor, the world will change once again.

CONCLUSION

Applying the Python language to digital investigation and forensic applications has great promise. What is needed is a collaborative community that includes: practitioners, researchers, developers, professors, students, investigators, examiners, detectives, attorneys, prosecutors, judges, vendors, governments, and research institutes. In addition, a cloud-based computing platform with thousands of cores, petabytes of storage, and terabytes of memory is necessary.

I challenge you to participate. If everyone reading this book would submit one idea, challenge problem or solution that would be a tremendous start. Visit Python-Forensics.Org to get started.... I would love to hear from you!

Additional Resources

Fighting Cyber Crime hearing before the subcommittee on Crime of the Committee on the Judiciary, http://commdocs.house.gov/committees/judiciary/hju72616.000/hju72616_0f.htm; 2001.

Forensic Information Warfare Requirements Study F30602-98-C-0243. Final technical report, February 2, 1999, Prepared by WetStone Technologies, Inc.
A road map for digital forensic research, Utica, New York, http://www.dfrws.org/2001/dfrwsrm-final.pdf; August 7–8, 2001.
January 1, 2014, http://www.statisticbrain.com/twitter-statistics/.

Cracking the 1,000-core processor power challenge. Nick Heath, http://www.zdnet.com/ cracking-the-1000-core-processor-power-challenge-7000015554/.

Index

Note: Page numbers followed by b indicate boxes, f indicate figures and t indicate tables.

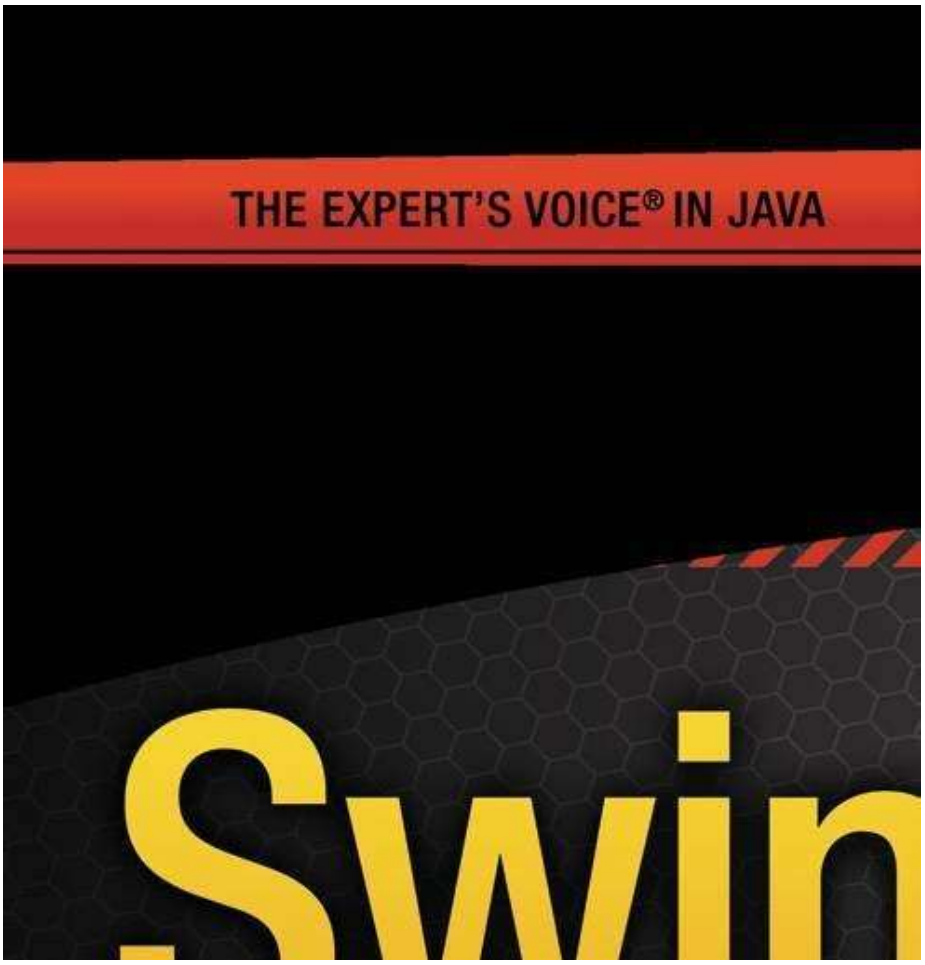
D

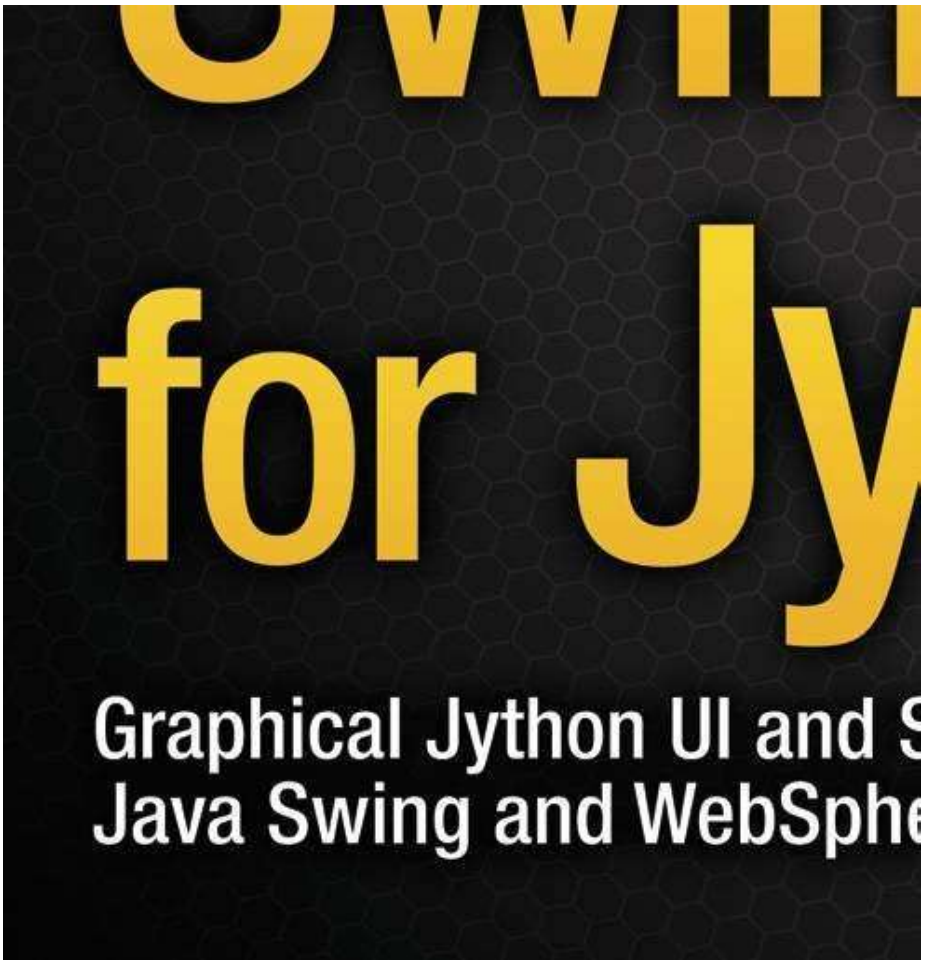
F

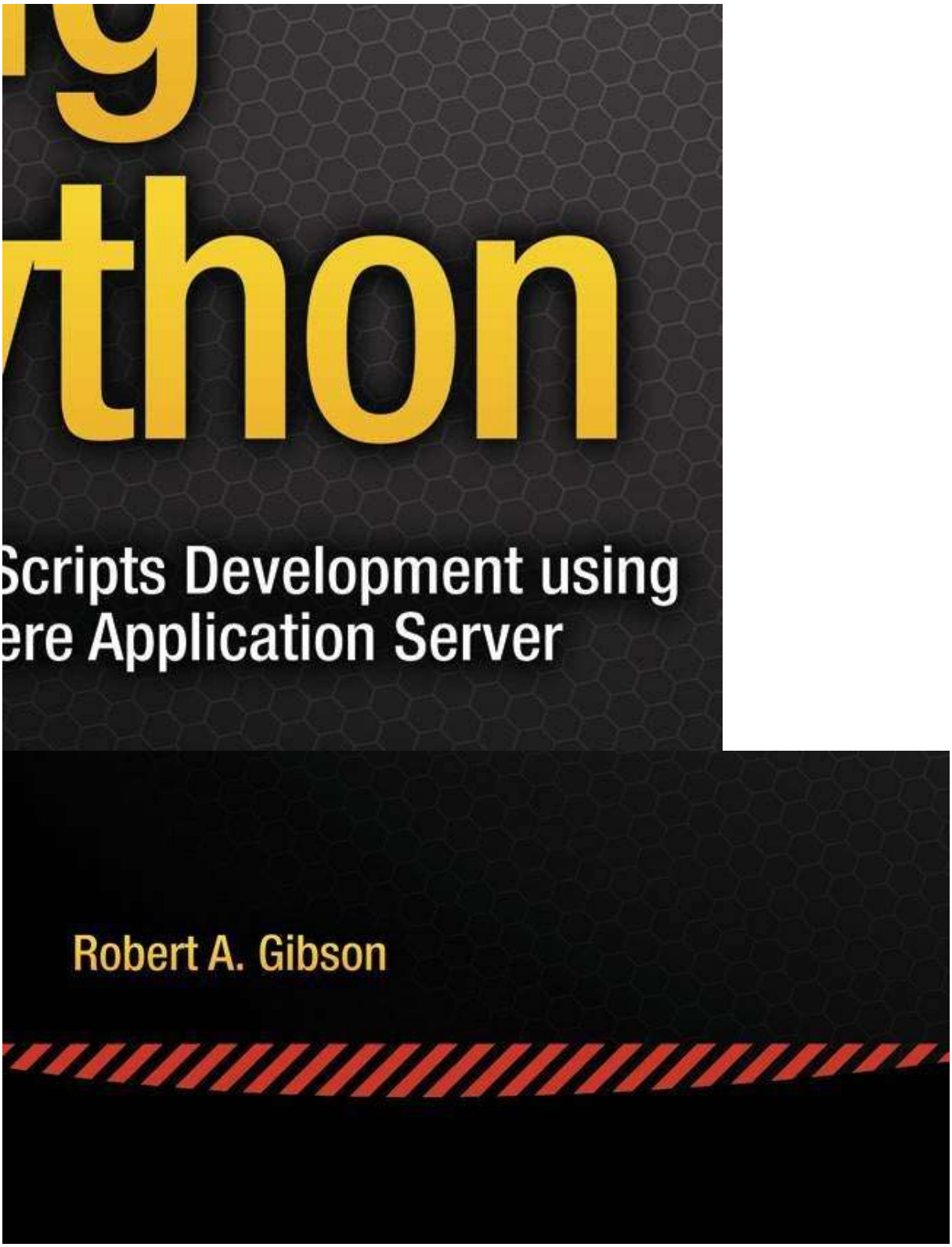
G

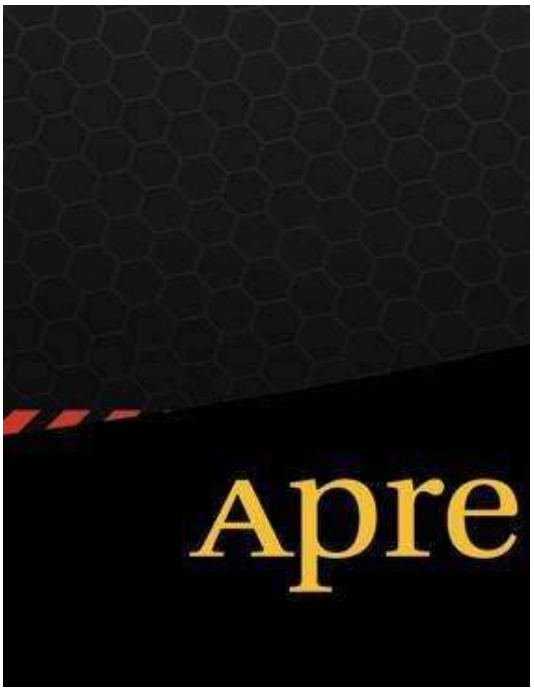
N

O




for Jy
Graphical Jython UI and S
Java Swing and WebSphe


g
thon
Scripts Development using
ere Application Server
Robert A. Gibson


Apre


SS®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Contents at a Glance

Introduction

A long time ago, in a galaxy far, far away...

Okay, maybe not so long ago, unless you are thinking in terms of “web years.” In 1995, Java was introduced into the world. At that time, it included a Graphical User Interface (GUI) framework called the Abstract Window Toolkit (AWT). Unfortunately, the AWT contained many native widgets that depended on the underlying operating system.

It didn’t take long to realize that the AWT approach was limited, and unfortunately, unreliable. So, in 1997, the Java Foundation Classes (JFC) was introduced and included the Swing component classes.[1]
Late in 1997, Jython was created to combine the performance provided by the Java Virtual Machine (JVM) with the elegance of Python. In so doing, the entire collection of existing Java classes were added to the ones available from Python. What this means to a software developer is that the rapid prototyping and iteration of Python can be combined with the impressive class hierarchy of Java to provide an amazing software development platform.
In 2002, Samuele Pedroni and Noel Rappin wrote a book titled *Jython Essentials* which, interestingly enough, uses an example similar to the one shown in Listing 1.

Listing 1. Welcome to Jython Swing

```
wsadmin>from javax.swing import JFrame
wsadmin>win = JFrame( "Welcome to Jython Swing" ) wsadmin>win.size = (
400, 100 )
wsadmin>win.show()
```

The output of this interactive wsadmin code snippet is shown in Figure 1.

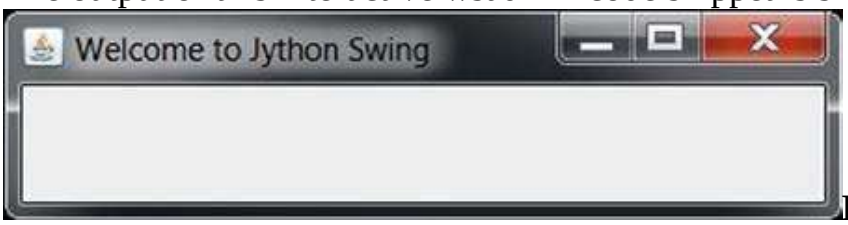


Figure 1. *Welcome to Jython Swing application output*

At the time of this writing, the first chapter of Jython Essentials was (still) available on the O’Reilly website.[2] So, it has been obvious, at least to some people, just how valuable and powerful Jython can be as a Swing development environment.

[1] I hadn’t realized, at least until I looked at *Java Foundation Classes in a Nutshell* by David Flanagan, just how much of the JFC was composed of Swing classes and components. Most of Part I of that book documents the GUI Application Programming Interfaces (APIs) used in client-side Java programs.
[2]See http://oreilly.com/catalog/jythoness/chapter/ch01.html.

Why Read This Book?

You may ask, if the *Jython Essentials* book has been around for more than a decade and talks about using Swing with Jython, is this book really necessary? The answer to that question is yes, because *Jython Essentials*, as well as the other Jython books, talk a little about using Swing with Jython and provides occasional example programs, but this topic is mentioned only in passing.

There are some other books about Jython, most notably *The Definitive Guide to Jython: Python for the Java Platform* by Jim Baker, Josh Juneau, Frank Wierzbicki, Leo Soto, and Victor Ng (Apress, ISBN 978-1-4302-2527-0). It too has some examples of using Swing with Jython. Unfortunately for the person interested in learning how to write Jython Swing applications, the amount of information is limited.

What Does This Book Cover?

The focus of this book, on the other hand, is to show you how to use Swing to add a GUI to your Jython scripts, with an emphasis on the WebSphere Application Server wsadmin utility. In fact, we teach you Swing using Jython and do it in a way that will make your scripts easier for people to use, more robust, more understandable, and therefore easier to maintain.

The Swing hierarchy is a huge beast.
How do you eat an elephant? One bite at a time.
That's what you're going to do with Swing in this book—consume it in a lot of small bytes.
In order to make it more easily consumable, the book uses lots of examples, with most of them building on earlier

ones. In fact, by the time you're done, you'll have touched on the almost 300 scripts that were written during the creation of this book.

Additional challenges exist, for example, event handling and threads. These too require some clarifying examples and explanation. We will also be dealing with concurrency, especially in the context of using threads in the applications that are created.

As you progress, you'll see that there are often a number of different ways to do

things. We try to point some of these different approaches, depending on the context. Why only some? Well, unfortunately, we are rarely able to identify them all. As I found after reading lots of different programs, there is often yet another way to do something. So, we admit that we don't know everything. In fact, that's one of the things that we find neat and interesting about writing code. We love to learn and hope that you do too.

Swing development is a "target rich" environment.
These days, it's a challenge to find command-line only programs. Wherever you look, programs have a graphical interface and allow the users to use their mouse to make selections. Frequently, you can use the mouse to completely specify the information required by a program to perform the user-desired operations. How many times have you been able to use a mouse to make all of the selections from the displayed information? I bet you don't have to think too hard to come up with a number of examples of this kind of interaction.
Unfortunately, this has not been the case for most WebSphere Application Server (WSAS) administrative scripts. When using wsadmin to execute one of these WSAS administrative scripts, developers have been forced to do one of the following:

• Provide command-line options as input.
• Use some kind of input file (such as a properties file, Windows .ini file, and so on).
• Have the script prompt the users and wait for them to provide an appropriate response.

This book is going to help you change all that. We're going to cover all of the information that you need to help you add a Graphical User Interface (GUI) to your WSAS Jython scripts. Does that mean that we cover each and every Java Swing class, method, and constant? No, unfortunately not. Take a look at *Java Swing* by Robert Eckstein, Marc Loy, and Dave Wood (ISBN 1-56592-455-X); it's more than 1,000 pages long! And, it's not the only huge book on Java Swing. Unfortunately, this is part of the problem. Many people are intimidated by the amount of information and are unsure of how and where to start.

One thing that you should realize is that we don't have to create a huge tome about each and every aspect on this subject in order to make it useful. For one,

we don’t duplicate information that is available elsewhere. What we do need to do is show you:

• What is possible
• What is required
• How to make use of existing information

• How to take Java examples and produce equivalent (possibly even better) Jython scripts that do the same kind of thing
And that is what we intend to do with this book. How does that sound?

What You Need

What is required?
This book is all about using the Java Swing classes in your Jython scripts. The fact that a number of examples use
the IBM WebSphere Application Server (WSAS) to demonstrate different things does not mean that you must have
WSAS in order to use Swing in your Jython scripts. I happen to use the WSAS environment to demonstrate some of
the more complete applications. So it is important to note that some, but not all, of the scripts included with this work
depend on information that is provided by a WSAS environment. If you are interested in using the information in this
book in your own Jython scripts, I encourage you to do so.
All of the book’s scripts have been tested using WSAS versions 8.5, 8.0, and 7.0. Some of them are also usable on
version 6.1, but there are some things that don’t exist in that version of wsadmin.[3] When these issues pop up, they
are addressed.
Most contemporary software programs have a graphical user interface. In fact, some people (like my kids) would
be stymied by something like a Windows or UNIX command prompt.
They would likely ask something like, “What am I supposed to do now? There’s nothing to click on!” That’s what
this book is all about—helping you create user-friendly Jython scripts using Java Swing classes.

[3]Most notably, any scripts that depend on the SwingWorker class won’t work on

version 6.1 of WSAS since that class is not available in the 6.1wsadmin class hierarchy.

Components and Containers

Before you begin your exploration of Swing objects and classes, I need to first explain how I am going to describe these things. For the most part, the objects that you use on your graphical applications are called *components*. In some places, they may be referred to as widgets. I'll try to be consistent and stick with components in hope of minimizing confusion.

In this chapter, you get your first exposure to the Swing hierarchy and see how a Jython application can use the Swing classes to create a Graphical User Interface (GUI). You'll begin with top-level container types and see what makes them special. Then you will see how Jython can help you to investigate and understand the class hierarchy. Next, you create a trivial application using an interactive session. You also get your first exposure to some of the challenges associated with the positioning of components on an application when you aren't aware of class default values. Finally, you'll see how users can impact the way that things are displayed should they resize the application window.

Another thing that you need to realize is that the Java Swing classes are not a complete replacement of the AWT. There are a number of places, as you will see, where AWT features continue to be used. For example, the AWT eventhandling elements and mechanisms are an integral part of the user interface that most people consider "Java Swing." When AWT features are needed, they are identified accordingly.

As you will soon see, you will be building applications using Swing components placed in a way to convey information to the users. Sometimes the users can interact with these components in order to provide information to the application. At other times, the components are used only to provide information to the users. An example of this kind of component is text placed on the application near an input field to direct users as to what kind of information, or input, is expected.

Sometimes, multiple components or objects are grouped together and associated with one another. An example of this is a list of some sort that's used to make a

selection. This grouping of components will be associated with, and contained within, a collection. One of the many concepts explained in this book is how to tell the Swing classes how a collection should be displayed. In fact, in order for a component to be visible, it must be associated with a container of some sort. Because of this, a hierarchy of containers and components exists in every Swing application. At the top of the hierarchy there needs to be a root, or top-level, container that holds the complete collection of application components.[1] This might make a little more sense when you see some examples.

Top-Level Containers

Some Swing containers are special. The main difference between these and other containers in the Swing hierarchy is that none of the top-level containers are descended from the javax.swing.JComponent class. In fact, each of them is a descendant of an AWT class. Because of this, these containers may not be placed into any other container. All of these special containers are called “top-level” containers because they are at the top (relatively speaking) of the application hierarchy.

[1]The biggest difference between collections and components is that a collection is a kind of component that can hold other components.

What are these container classes, and what does it mean that they aren’t descended from javax.swing. JComponent?
Take a look at the Java class documentation (i.e., the output of the Javadoc tool that’s used to generate API documentation in HTML).[2] In this documentation, you need to locate the top-level containers. Table 1-1 shows the top portion of the class hierarchy for each of the top-level containers.

Table 1-1. *Top-Level Containers* **Container Type** JApplet[3]

JDialog[4]
JFrame[5]
JWindow[6]
Class Hierarchy

java.lang.Object java.awt.Component java.awt.Container java.awt.Panel java.applet.Applet javax.swing.JApplet java.lang.Object java.awt.Component

java.awt.Container java.awt.Window
java.awt.Dialog
javax.swing.JDialog java.lang.Object java.awt.Component java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame java.lang.Object java.awt.Component java.awt.Container
java.awt.Window
javax.swing.JWindow

As you can see, each of these classes descends from a java.awt class, not from the javax.swing.JComponent class. As with all AWT classes, this means that there is a significant portion of the class that is composed of native (i.e., operating system-specific) code.

[2] See http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.
[3]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JApplet.html.
[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JDialog.html.
[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.
[6]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JWindow.html.

Looking at these top-level components might make you wonder about the difference between a window (JWindow) and a frame (JFrame). Listing 1-1 shows a trivial interactive wsadmin session[7] that can be used to display a JWindow instance of a specific size.[8]

Listing 1-1. Simple JWindow example

```
wsadmin>from java.awt import Dimension
wsadmin>from javax.swing import JWindow
wsadmin>win = JWindow()
wsadmin>win.setSize( Dimension( 400, 100 ) )
wsadmin>win.show()
```

Wait a minute. How did I know that I needed to import the Dimension class from the java.awt library and then instantiate one of them in order to invoke the setSize() method?
The answer is I cheated. I first tried to do it without importing the Dimension class, as in Listing 1-2.

Listing 1-2. The setSize() exception

```
wsadmin>from javax.swing import JWindow
wsadmin>win = JWindow()
wsadmin>win.setSize( ( 400, 100 ) )
WASX7015E: Exception running command: "win.setSize( ( 400, 100 ) )";
exception information: com.ibm.bsf.BSFException: exception from Jython: ...
setSize(): 1st arg can't be coerced to java.awt.Dimension
```

Did you notice how the exception tells you exactly what you need to use to resolve the issue? This demonstrates just how easy it is to use an interactive wsadmin (or Jython) session to develop and test your applications.[9]
Now, getting back to the JWindow. If you execute the steps shown in Listing 1-1, you'll notice how empty it is. It is a completely blank slate. This provides you with the opportunity to completely define how your application will look. The tradeoff though is that you have to define each and every aspect of the application. For this reason, however, I prefer to use the JFrame as a starting point (at least for now), since it does a lot of the work for me. In fact, the vast majority of scripts in the remainder of the book use the JFrame class.

Getting Help from Jython

You just looked at the Java class documentation for the top-level containers. Do you really have to use the documentation, or is there any way to get to this kind of information from Jython? Let’s take a quick look at what Jython can do to help you. Listing 1-3 shows an interactive wsadmin session that includes the definition of a simple Jython class function (called classes) that can display information about the class definition hierarchy. In this case, this function is used to show information about the JFrame class.

[7] To start an interactive session, execute either the wsadmin.bat or wsadmin.sh shell script and identify the scripting language to be used as Jython. For example, "./wsadmin.sh -conntype none -lang Jython".
[8]I did not include a figure of the result because it is simply a plain white rectangle displayed in the top-left corner of the screen. Go ahead and try it for yourself, and you’ll see what I mean.
[9]You'll find that we often use script and application interchangeably in this book.

Listing 1-3. The classes function

```
wsadmin>from javax.swing import JFrame wsadmin>
wsadmin>def classes( Class, pad = '' ) : wsadmin> print pad + str( Class )
wsadmin> for base in Class.__bases__ : wsadmin> classes( base, pad + '| ' )
wsadmin>
wsadmin>classes( JFrame )
javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible | | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

How does this information compare with what you saw earlier from the Java class documentation? One difference is that the information that is displayed is from the specified class, which in this case is JFrame, down toward its ancestor components. It is also more complete. The really neat thing about this is that you are using the power of Jython to understand what is available from the Java Swing hierarchy.

How and Why Are You Able to Do This?

Java includes some properties that allow objects and classes to be dynamically "queried" to determine information about what the objects and classes can do. These properties are called *reflection* and *introspection*. The classes function in Listing 1-3 illustrates one way to use these properties. I won't get into this too much, at least at this point, but you will be seeing more about how to use some of the properties later.[10]

What's Next?…Starting Simple

After starting with a frame, you need to decide what your application should display. Let’s start by adding the ubiquitous "Hello World!" message. How do you go about doing that? First, you have to figure out what kind of object you need to display text.

Swing has a component for that, called a JLabel. (You will soon realize that many of the Java Swing components start with a capital J).
[10]One important point to mention is that the output produced by the classes function when Jython (not wsadmin) is used is much harder to read. I chose to use the wsadmin environment for simplicity and readability.

Before you create a JLabel object, you need to tell Jython where it can obtain details about this class, just like you did for the JFrame class. Again, you use a variation of the Jython import statement. Listing 1-4 shows one way to do exactly this.

Listing 1-4. Adding a JLabel to the application

```
wsadmin>from javax.swing import JFrame
wsadmin>from javax.swing import JLabel
wsadmin>
wsadmin>frame = JFrame( 'Hello world' )
wsadmin>label = frame.add( JLabel( 'Hello Swing world' ) )
wsadmin>frame.pack()
wsadmin>frame.setVisible( 1 )
wsadmin>
```

The steps used in this example can be described as follows:
1. Instantiate (create) a JFrame object (supply the title). This is done by calling the JFrame constructor.
2. Instantiate a JLabel object (supply the label text). This is done by calling the JLabel constructor.
3. Add the JLabel object to the JFrame (application) object. This is done by calling the JFrame add() method and passing the JLabel object to it.[11]
The result of executing the code in Listing 1-4 is a small application window located in the top-left corner of the screen. first image in Figure 1-1

Figure 1-1. *The*

“Hello Swing world” window

The shows what this window looks like. It’s interesting to note, however, that the application title, which is normally on the title bar, is obscured by the application icons. The second image in Figure 1-1 shows what happens when you grab the right side of the window and drag it to the right, thereby increasing the window width and making the application title visible.

Adding a Second Label

Hopefully, that seems pretty straightforward to you. Let’s add another label just to see what happens. Listing 1-5 shows a different interactive wsadmin session that does just that.

[11]The program saves the result of calling frame.add() into a variable simply to make the interactive session more readable. **Listing 1-5.** Adding a second JLabel to the application

```
wsadmin>from javax.swing import JFrame
wsadmin>from javax.swing import JLabel
wsadmin>
wsadmin>frame = JFrame( 'Hello world' )
wsadmin>label = frame.add( JLabel( 'Hello Swing world' ) ) wsadmin>label =
frame.add( JLabel( 'Testing, 1, 2, 3' ) ) wsadmin>frame.pack()
wsadmin>frame.setVisible( 1 )
wsadmin>
```

Figure 1-2 shows the output. Unfortunately, it probably doesn’t look like you expected it to. What happened? The simple answer is that you didn’t tell the frame where to add the second label, so it put both labels in the same place, and Swing can’t show both labels in the same place. This chapter isn’t the best place get into details about Layout Managers; you’ll learn about them in Chapter 5. You’ll do just enough to get by here.

Figure 1-2. *After adding the second JLabel*

Is there a something simple that you can do make this example work? Yes there

is! Listing 1-6 shows how you can change the default Layout Manager used by the JFrame objects.[12]

Listing 1-6. Changing the default JFrame Layout Manager

```
wsadmin>from java.awt import FlowLayout
wsadmin>
wsadmin>from javax.swing import JFrame
wsadmin>from javax.swing import JLabel
wsadmin>
wsadmin>frame = JFrame( 'Frame title' )
wsadmin>frame.setLayout( FlowLayout() )
wsadmin>label = frame.add( JLabel( 'Hello Swing world' ) )
wsadmin>label = frame.add( JLabel( 'Testing, 1, 2, 3' ) )
wsadmin>frame.pack()
wsadmin>frame.setVisible( 1 )
wsadmin>
```

Unfortunately, as you can see in the image on the left in Figure 1-3, the two labels are side by side on the same line. If you drag the corner of the window to narrow the application window a little bit, you'll get better results. The image on the right in Figure 1-3 shows how the two labels are separate and distinct.

[12]In case you are interested, BorderLayout is the default Layout Manager used by JFrame.

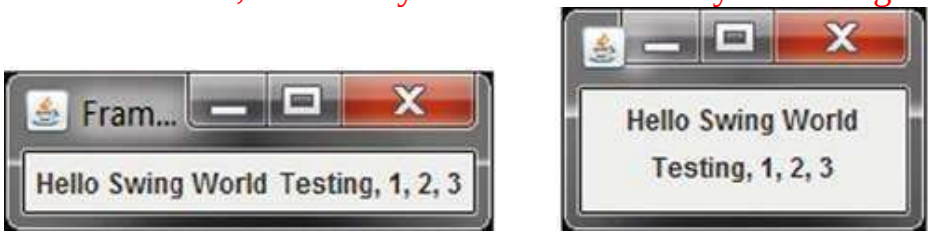


Figure 1-3. *Adjacent JLabel objects*

Summary

What has this chapter taught you? For one thing, it shows how easily you can create a trivial graphical application using Jython and Swing. However, these aren't really good examples because they were created using interactive wsadmin sessions. In the next chapter, you'll learn about the differences between interactive sessions and script files.

Chapter 2

Interactive Sessions vs. Scripts

It shouldn’t take long for you to realize that you don’t want to be using interactive wsadmin sessions for your applications. What does it take? Well, there are some things of which you must be aware.

This chapter begins with some additional interactive scripts to help illustrate some of the important differences between the interactive environment and what is needed for your Jython Swing scripts to be successful. Part of this process includes using the Java compiler to understand that some methods have been deprecated. Unfortunately Jython doesn't warn you about this when the scripts using those methods are executed. Finally, you’ll take a look at thread safety and the challenge that it presents to developers.

Running Your First Script from a File

Let’s start by putting the trivial script from the previous chapter into a text file and then execute it using wsadmin.[1] Listing 2-1 shows the contents of the Welcome.py script file.

Listing 2-1. The Welcome.py Script File

```
from javax.swing import JFrame
win = JFrame( 'Welcome to Jython Swing' ) win.size = ( 400, 100 )
win.show()
```

What happens when you execute this script using wsadmin?[2] Nothing, that’s what. The question is, why don’t you see anything? The simple answer is that the call to the show() method returns immediately, and the script exits. There isn’t time for the Swing framework to display the instantiated window. To verify this, you can add a statement that causes the script to wait. The easiest and simplest way to do this is to use the raw_input() function to display a message and then wait for user input. Listing 2-2 shows the contents of the modified Welcome1.py script file.

Listing 2-2. The Welcome1.py Script File

```
from javax.swing import JFrame
```

```
win = JFrame( 'Welcome to Jython Swing' )
win.size = ( 400, 100 )
win.show()
if 'AdminConfig' in dir() :

raw_input( '\nPress <Enter> to terminate the application: ' )
```

[1]For those using Jython and not wsadmin, you don't need to use raw_input() to pause the script. [2]The command should look something like wsadmin -conntype none -f Welcome.py.

That's better! When you execute this script, the wsadmin utility stays around long enough for the application window to be displayed. Even though the application isn't too exciting, it is interesting enough as a starting point.
The next question you should ask at this point is, "What happens when you use (click on) the application close icon in the top-right corner of the application window?" The application exits, right? No, it doesn't. Look at the interactive wsadmin command prompt window. It continues to show the "Press <Enter> to terminate the application:" message. If you press the Enter key at this point, wsadmin exits and the operating system command prompt is displayed.[3]
How do you fix this behavior? How do you get wsadmin to exit when you use the application's close icon? The simple answer is that you have to tell it to. If you take a moment to think about it, you'll realize that the close icon is part of the JFrame. If you take a look at the JFrame online documentation,[4] you'll find the following:

Unlike a Frame, a JFrame has some notion of how to respond when the user attempts to close the window. The default behavior is to simply hide the JFrame when the user closes the window. To change the default behavior, you invoke the setDefaultCloseOperation(int) method.

Listing 2-3 shows the revised script, Welcome2.py. The only change from the previous script is the addition of a call to the setDefaultCloseOperation() method. What happens when you run this script and click on the close icon? The application window is removed, and the wsadmin utility terminates.

Listing 2-3. The Welcome2.py Script File

```
from javax.swing import JFrame
win = JFrame( 'Welcome to Jython Swing' )
```

```
win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
win.size = ( 400, 100 )
win.show()
if 'AdminConfig' in dir() :

raw_input( '\nPress <Enter> to terminate the application: ' )
```

This is perfect, right? No, not really. There are a couple of things that aren't as they should be.

Depending "Too Much" on Limited Information

The first problem with the trivial script in Listing 2-3 isn't very obvious. In fact, to figure out this problem, you can either:

- Look closely at the JFrame documentation.
- Write and compile an equivalent Java application.

Listing 2-4 shows an equivalent Java application.

[3]For readers who use Jython and not wsadmin, the Java process will still be executing in the background when you execute the script. [4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.

Listing 2-4. Welcome3.java import javax.swing.JFrame;

```
public class Welcome3 {
public static void main( String args[] ) {
JFrame win = new JFrame( "Welcome to Java Swing" );
win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); win.setSize( 400,
100 );
win.show();

}
}
```

What happens when you compile this? The warning messages (notes) shown in Figure 2-1 are generated.

```
Note: Welcome3.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

Figure 2-1. *Welcome3 warning messages*

If you recompile Welcome3 using the specified option, you get a more detailed explanation of the problem, as shown in Figure 2-2.

```
warning: [deprecation] show() in java.awt.Window has been deprecated
```

Figure 2-2. *Welcome3 detailed deprecation message*

Looking at the documentation for this show() method,[5] you'll find that you should be using setVisible(boolean) instead.
This shows[6] you a challenge that you'll encounter when using Jython to call Java methods. The Java compiler informs the users when a method has been deprecated, but the Jython environment does not. I'm not suggesting that you should avoid using calls to Java methods in your Jython scripts. Far from it' I'm just letting you know that you should check to see if a method has been deprecated before using it in your scripts. This situation is most likely to occur, at least from my experience, if you obtain an existing Java Swing program and translate, or convert, it to the equivalent Jython without checking for this kind of issue.

[5]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/Window.html#show%28%
[6]Pun intended.

Swing Threads

Deprecated methods aren't the only kind of issue of which you need to be aware. One of the most significant differences between simple applications and ones where the developer needs to be able to interact with users and respond to events is related to threads of control (aka threads). Articles have been around for quite some time that discuss issues related to the fact that Swing developers should be aware of the fact that most Swing components are not thread-safe,[7] and techniques exists for performing long-running operations.[8]

One technique is to define the application in a separate class and then wrap the instantiation of this class in a Java Runnable class. The calling of this Runnable class is deferred until the Swing environment is ready for it. This slight delay is performed by the Swing Event Dispatch thread and is initiated by a call to the invokeLater() method call of the SwingUtilities, or EventQueue class. This concept can be hard to follow. Take a look at an example of this technique, as shown in Listing 2-5.

Listing 2-5. Template1.py

```
1|import java
2|import sys
3|from java.awt import EventQueue
```

```
4|from javax.swing import JFrame
5|class Template1 :
6| def __init__( self ) :
7| frame = JFrame( 'Title' )
8| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
9| frame.pack()
10| frame.setVisible( 1 )
11|class Runnable( java.lang.Runnable ) :
12| def __init__( self, fun ) :
13| self.runner = fun
14| def run( self ) :
15| self.runner()
16|if __name__ in [ '__main__', 'main' ] :
17| EventQueue.invokeLater( Runnable( Template1 ) )
18| if 'AdminConfig' in dir() :
19| raw_input( '\nPress <Enter> to terminate the application: ' ) 20|else :
21| print '\nError: This script should be executed, not imported.\n' 22| if
'JYTHON_JAR' in dir( sys ) :
23| print 'jython %s.py' % __name__
24| else :
25| print 'Usage: wsadmin -f %s.py' % __name__
26| sys.exit()
```

A detailed description for this code can be found in Table 2-1. [7]See http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html. [8]See http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html.

Table 2-1. *Template1.py Details*

Lines	Detailed Description
1-4	Statements used to add class names to the Jython namespace.
5-10	User-defined application class (i.e., Template1).
11-15	Wrapper class, descended from java.lang.Runnable, that saves a class reference in the constructor and delays the instantiation of the class until the run() method is called (on the Swing Event Dispatch thread).
16-22	This is the (apparent) script-entry point. This code determines if the script was executed or imported. If imported, an error message is displayed and the script exits. If the script was executed, a call to instantiate the user application

class is deferred until the Swing Event Dispatch thread is ready to do so.

A roughly equivalent approach is shown in Listing 2-6.
Listing 2-6. Template2.py

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from javax.swing import JFrame
5|class Template2( java.lang.Runnable ) :
6| def run( self ) :
7| frame = JFrame( 'Title' )
8| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
9| frame.pack()
10| frame.setVisible( 1 )
11|if __name__ in [ '__main__', 'main' ] :
12| EventQueue.invokeLater( Template2() )
13| if 'AdminConfig' in dir() :
14| raw_input( '\nPress <Enter> to terminate the application: ' ) 15|else :
16| print '\nError: This script should be executed, not imported.\n' 17| if
'JYTHON_JAR' in dir( sys ) :
18| print 'jython %s.py' % __name__
19| else :
20| print 'Usage: wsadmin -f %s.py' % __name__
21| sys.exit()
```

Looking closely at these two listings, you should notice the differences. Instead of defining a separate user application class that creates the Swing application components in the class constructor, this code places all of the Swing component-creation operations in the class run() method.

The example scripts provided and described in this book tend to use the second template since it requires a little less code. It also makes a bit more sense, at least to me. While writing the example scripts in this book I found it extremely easy to start with the Template script and add the code that was required to demonstrate the topic being discussed. One useful feature of this technique is the fact that it should help you focus on the important differences and hopefully spend less time with the script as a whole.

Summary

This chapter shows what happens when you start using script files instead of interactive wsadmin sessions for your applications. The biggest difference is that you have to add a way for the script file to wait for the users to interact with the application. To do so, the script files will often use something like the raw_input() function provided by Jython. Additionally, subsequent script files in this book include an application class that defines the Jython components, containers, and structures used by the applications demonstrating the use of Swing classes and constructs.

Chapter 3

Building a Simple Global Security Application

In my experience, learning a programming topic is much easier when good examples are included. Throughout this book, you're going to go through the process of building simple Jython Swing applications from scratch. As you do so, you will be learning about some of the available Swing components and using them to make your application useful. In this chapter, you take a quick look at a simple graphical application that displays whether the WebSphere Global Security has been enabled or not. This will lead you to the topics of panes and layers, which are so very important to Swing applications.

Adding Text to the Application Using a JLabel

At this point, you haven't learned how to build a non-trivial, fully functional application. However, you do have information eno ugh to get started. How so? Well, you'll start simple by building a simple application that displays the status of the global security. For those who may be unfamiliar with it, global security is simply a setting that determines whether a username and password is required to administer the application server environment. Listing 3-1 shows a complete Jython script to do just this.

Listing 3-1. SecStatus.py

```
1|import java
2|from java.awt import EventQueue
3|from javax.swing import JFrame
```

```
4|from javax.swing import JLabel
5|class SecStatus( java.lang.Runnable ) :
6| def run( self ) :
7| frame = JFrame( 'Global Security' )
8| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
9| security = AdminConfig.list( 'Security' )
10| status = AdminConfig.showAttribute( security, 'enabled' ) 11| frame.add(
JLabel( 'Security enabled: ' + status ) ) 12| frame.pack()
13| frame.setVisible( 1 )
14|if __name__ in [ '__main__', 'main' ] :
15| EventQueue.invokeLater( SecStatus() )
16| raw_input( '\nPress <Enter> to terminate the application: ' ) 17|else :
18| print 'Error: This script should be executed, not imported.\n' 19| print 'Usage:
wsadmin -f %s.py' % __name__
```

The statements that make use of wsadmin scripting objects are shown in lines 9 and 10. This application doesn't require a graphical user interface. In fact, it can easily be performed using a single wsadmin command with a print statement to display the status. Listing 3-2 shows a simple wsadmin command line that can be used to display the same security status information.[1]

Listing 3-2. wsadmin Command that Displays the Global Security Status

```
wsadmin -conntype none -lang jython -c "print 'Security enabled: ' +
AdminConfig.showAttribute( AdminConfig.list( 'Security' ), 'enabled' )"
```

However, you will quickly see that scripts that interact with users are often good candidates for graphical user interface, and so that is what you will be learning to do.

No "Pane," No Gain

When your application creates one of the top-level containers (e.g., a JFrame), a number of things are created by that container to help it perform its role. One of the things created by the JFrame constructor is a RootPane (which happens to be of type JRootPane).[2]

What does this RootPane do? It is composed of, and is used to manage, the following panes, each of which will be described in the following sections:

- A glass pane
- A layered pane

• A content pane
• An optional menu bar

When You Live in a Glass House, Everything Is a Pane

The glass pane can be used to intercept events.[3] For most applications, it is a transparent pane that doesn't get in the way of your visible application components. However, there may be a time when you want to have something made visible on the glass pane that covers an existing component on the content pane. There is an interesting example in the *Java Swing Tutorial*, on the :How to Use Root Panes" page,[4] that demonstrates one possible use of the glass pane. A section about the glass pane can be found on that same page.[5]

At this point, I haven't discussed enough of using Swing with Jython for the novice user to understand this particular example. However, someone with some knowledge of Java Swing and Jython might be interested in how an equivalent Jython script would look. For those folks, I have included a translated version of the GlassPaneDemo.py script to read and play with. Figure 3-1 shows some sample images that can be created using this script. The complete script is available in the source code for this chapter.

[1] The contents of Listing 3-2 must be entered as a single line / statement.
[2]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JRootPane.html.
[3]Which have not yet been defined. For the time being, you can think of things like mouse clicks as events, the first discussion of which appears in Chapter 4.
[4]See http://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html.
[5]See http://docs.oracle.com/javase/tutorial/uiswing/components/rootpane.html#glasspa

Figure 3-1. *GlassPaneDemo.py sample images*

The Layered Look Can Also Be a Pane

I'm certain that you have used an application that includes components that involve different layers. One of the most common instances of layers occurs

when a pop-up menu is displayed. It is important that this menu isn't obscured by any of the existing components. The layered pane is used to great effect for component positioning. Right now, you don't have to worry about this pane. It is discussed in Chapter 19.

The Optional MenuBar

Almost every graphical application that you can think of has some kind of menu. The MenuBar is a collection of MenuItems that are displayed to allow users to make selections. The applications that you create are likely to use MenuBars. In fact, many of the complete scripts that are developed in the last few chapters of this book use MenuBars and MenuItems. Since this is an important topic, the book spends all of Chapter 10 on menu-related issues.

The Content Pane Will Contain Most of the Visible Items

All Swing applications have at least one top-level container. Each of these top-level containers is the top, or root, of the application containment hierarchy (i.e., all of the visible components that appear in the container). In this book, almost all of the example applications will have a JFrame as the top-level container.

Additionally, just about every application will place the components to be displayed on the JFrame content pane. So, one of the most common things that needs to be done with a JFrame instance is to obtain a reference to its content pane. This is typically done using syntax like that shown in Listing 3-3.

Listing 3-3. Getting a JFrame Content Pane Reference contentPane = frame.getContentPane()

What kind of object is returned by this call? The simple answer is that a java.awt.Container[6] is returned. What, exactly, is a java.awt.Container? It certainly is a lot easier to comprehend than it is to describe. In fact, I described Swing containers as components that can hold a group of zero or more components.

[6]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html#getConte

Looking at the documentation,[7] you can see that it has six dozen methods. Granted, seven of these have been deprecated and another five are protected,

which means that you really shouldn't be using them from your Jython scripts. Another seven have multiple overloaded variations (e.g., there are five different add() methods). These add() methods are shown in Table 3-1.

Table 3-1. *Component add() methods*

Modifier and Type	Method and Description
Component	add(Component comp) Appends the specified component to the end of this container.
	Component add(Component comp, int index) Adds the specified component to this container at the given position.
	void add(Component comp, Object constraints) Adds the specified component to the end of this container.
	void add(Component comp, Object constraints, int index) Adds the specified component to this container with the specified constraints at the specified index.
	Component add(String name, Component comp) Adds the specified component to this container.

I won't be digging too deeply into these various methods, but I will be making good use of a number of them. In fact, you have already seen some example uses of the add() method (see line 11 of Listing 3-1, for example).
Did you catch that? I snuck one in on you there. If you were watching closely, you may have wondered why Listing 3-1 doesn't include a call to the getContentPane() method, as shown in Listing 3-3. How does this work? Well, the Swing developers realized that some method calls are very common, so they have provided a convenience. Instead of writing a statement like you see in Listing 3-4, you are allowed to leave out the call to the getContentPane() method.

Listing 3-4. Calling getContentPane() to add() Something

```
frame.getContentPane().add( something )
```

Where is this documented? Right on the JFrame documentation,[8] where you'll find the paragraph shown in Figure 3-2.[9]

[7] See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html.

[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.
[9]Yes, convenience is misspelled in the documentation.

... As a conveniance add and its variants, remove and setLayout have been overridden to forward to the contentPane as necessary. This means you can write:
frame.add(child);
And the child will be added to the contentPane.
Figure 3-2. *JFrame conveniences from the JFrame documentation*

Can you still include the call to the getContentPane() method in your code? Certainly. You aren't forced to use this convenience. If this makes your code easier and more understandable, that is your choice. Don't be surprised when most of the sample Java Swing code found in the wild[10] uses this convenience.

Summary

In this chapter, you saw a simple Jython Swing application that displays the current WebSphere global security status as the text of a JLabel component. Even though it is trivial in nature, it demonstrates the structure of the applications to be found throughout the book. Additionally, you had your first exposure to the layering that exists in the JFrame class, which will be used in almost every example application found in this book.

[10]The "wild" here refers to the wild, wild west, also known as the World Wide Web (WWW). **Chapter 4**

Button Up! Using Buttons and Labels

Up to this point, the applications you built in this book have been kind of boring.[1] They really haven't done anything except display a message. But now it's time to change that. In this chapter, you're going to add a button to your application and then have it react when the user presses it. How does that sound?

To do this you'll first take a look at the JButton class hierarchy and see the Java way to create a button and add it to the application frame. Then you'll see a verbose way, in Jython, to do the same thing. Then you'll see a more concise way to put a button on the frame and have an event handler update another

component when the user presses the button. This technique will allow your applications to react when the users click one of the buttons.

JButton Class Hierarchy

Before you start making buttons, it seems appropriate to take a look at this class. In Chapter 1, you saw a function named classes that displayed the class hierarchy for a given type from the inside out. Listing 4-1 shows this simple function again.[2]

Listing 4-1. classes.py

```
def classes( Class, pad = '' ) : print pad + str( Class )
for base in Class.__bases__ :

classes( base, pad + '| ' )
```

Using this function, you can display the JButton class hierarchy. Listing 4-2 shows an interactive wsadmin session that does just that.[3]

Listing 4-2. JButton class hierarchy

```
wsadmin>from javax.swing import JButton wsadmin>
wsadmin>classes( JButton )
javax.swing.JButton
| javax.swing.AbstractButton
| | javax.swing.JComponent
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer | | | | |
java.io.Serializable
| | | java.io.Serializable
| | java.awt.ItemSelectable
| | javax.swing.SwingConstants
| javax.accessibility.Accessible wsadmin>
```

[1] Except, of course, for the GlassPaneDemo.py script mentioned in Chapter 3, which was converted from the original Java. But it doesn't count since I haven't covered the concepts used in that code.

[2]Remember that the output shown in this book is from the wsadmin version of Jython since it's easier to understand.

[3]The command used to start this wsadmin session included the -profile classes.py parameters.

What are you going to do with this information? Well, the first thing that you need to do is to create, or instantiate, a button instance. Then you'll add it to the frame. To begin, take a quick look at a trivial Java application to do this. Listing 4-3 contains the relevant code snippet.[4]

Listing 4-3. The Java Code to Create a Button and Add It to a Frame

```
JFrame frame = new JFrame( "ButtonDemo" );
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ); JButton button =
new JButton( "Press me" );
frame.add( button );
frame.pack();
frame.setVisible( true );
```

What would this look like in Jython? Listing 4-4 shows the equivalent Jython code.

Listing 4-4. The Verbose Jython Way to Do the Same Thing

```
frame = JFrame( 'ButtonDemo_01' )
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) button =
frame.add( JButton( 'Press me' ) )
button.addActionListener( ButtonPressed() )
frame.pack()
frame.setVisible( 1 )
```

They certainly do look similar, don't they? What if you use an interactive wsadmin session to execute these steps? Listing 4-5 shows what happens when you try to do this.

Listing 4-5. The wsadmin interactive session to create and add a button

```
wsadmin>from javax.swing import JFrame, JButton
wsadmin>frame = JFrame( 'ButtonDemo_01' )
wsadmin>button = JButton( 'Press me' )
wsadmin>frame.add( button )
javax.swing.JButton[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=...
```

[4]The complete application can be found in the code\Chap_04\ButtonDemo_01.java file.

The figure ends when some output is generated by the frame.add() method call. What was returned by the add() method call? Listing 4-6 shows a slightly different interactive session. This time, the value returned by the add() method call is saved into a variable named result. Then you determine if the value returned by add() is the same one passed in. Why? Well, the JFrame class[5] inherits five different add() methods from the java.awt.Container. This Jython statement is calling the Container add() method[6] that returns the same value that was passed in.

Listing 4-6. The wsadmin interactive JButton session 2

```
wsadmin>from javax.swing import JFrame, JButton wsadmin>
wsadmin>frame = JFrame( 'ButtonDemo' ) wsadmin>button = JButton( 'Press
me' ) wsadmin>result = frame.add( button )
wsadmin>result == button
1
wsadmin>
```

You can use this fact to simplify your code. Listing 4-7 shows this simplification. Instead of instantiating the JButton instance and saving the returned value in a variable (e.g., button), you can combine the add() call with the JButton constructor call and save the value returned from the add() call.

Listing 4-7. The wsadmin interactive JButton session 3

```
wsadmin>from javax.swing import JFrame, JButton
wsadmin>
wsadmin>frame = JFrame( 'ButtonDemo' )
wsadmin>frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
wsadmin>button = frame.add( JButton( 'Press me' ) ) wsadmin>...
```

■ **Note** this technique doesn’t work for every Container add() method call, since not every add method returns a value.

The application found in Button_01.py contains a button that says “Press me!” that can be pressed. Unfortunately, it doesn’t do anything yet, but you’ll get there

right after a short digression. At this point it is appropriate to address an important question: How does Swing know where to put things on the application?

The Layout of the Land

One of the many parts of a Container is something called the Layout Manager. I touched on this a little in Chapter 1, and I will be going into detail about it in Chapter 5. For the moment, you'll simply take a quick peek "under the covers," so to speak.

For the time being, you're going to use JFrame as the top-level container. One way that you can learn more about the JFrame class hierarchy is to use the Java documentation. Figure 4-1 shows this relationship, at least the portion covered by the Java documentation.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html.
[6]See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#add%28java.av
java.lang.Object
java.awt.Component
java.awt.Container
java.awt.Window
java.awt.Frame
javax.swing.JFrame
Figure 4-1. *JFrame class hierarchy*

The root of this class hierarchy is an Object, which isn't too surprising. The class hierarchy for the JFrame Swing class can be seen in Listing 4-8. Note, however, that unlike the JFrame Javadoc class hierarchy shown in Figure 4-1, the output of the classes function starts at the JFrame class and works its way up the hierarchy. So this is kind of an inside-out view of the hierarchy. One of the things that I found interesting about this view is the presence of additional classes that don't show up in the Javadoc.

Listing 4-8. Another look at the JFrame class hierarchy

```
wsadmin>from javax.swing import JFrame wsadmin>
wsadmin>classes( JFrame )
```

```
javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

This diagram is kind of nice, but it would be more useful if it also allowed you to see the methods and attributes that exist at each level of the hierarchy. Better yet, it would be really nice if you had a way to filter the list of methods and/or attributes based on some text. Listing 4-9 contains a function that includes these additional capabilities. **Listing 4-9.** classInfo.py

```
1|def classInfo( Class, meth = None, attr = None, pad = " ) :
2| print pad + str( Class )
3| prefix = pad + ' '
4| if type( meth ) == type( " ) :
5| comma, line = ", " + prefix
6| methods = [
7| n for n, v in vars( Class ).items()
8| if n.lower().find( meth.lower() ) > -1 and callable( v )
9| ]
10| methods.sort()
11| for m in methods :
12| if len( line + comma + m ) > 65 :
13| print line.replace( '|', '>' )
14| comma, line = ", " + prefix
15| line += comma + m
16| comma = ', '
17| if not line.endswith( ' ' ) :
18| print line.replace( '|', '>' )
```

```
19| if type( attr ) == type( '' ) :
20| comma, line = ', ' + prefix
21| attribs = [
22| n for n, v in vars( Class ).items()
23| if n.lower().find( attr.lower() ) > -1 and not callable( v ) 24| ]
25| attribs.sort()
26| for a in attribs :
27| if len( line + comma + a ) > 65 :
28| print line.replace( '|', '*' )
29| comma, line = ', ' + prefix
30| line += comma + a
31| comma = ', '
32| if not line.endswith( ' ' ) :
33| print line.replace( '|', '*' )
34| for b in Class.__bases__ :
35| classInfo( b, meth, attr, pad + '| ' )
```

How do you use it? Well, when you call the classInfo() function, you must specify at least one parameter, which is a class (e.g., JFrame). In addition, you can als provide one or two parameters to show the methods and/or attribute names at each level of the class hierarchy. I find it best to include the parameter keyword (i.e., meth or attr) to indicate which parameter is being provided.

For example, if you want to display all of the methods in the JFrame hierarchy that contain the case-insensitive string "layout", you could call this function a statement, as shown on line 3 in Figure 4-7. Method names that include the specified text (i.e., "layout") are indicated using the > character.

Listing 4-10. JFrame class hierarchy showing the “layout” methods.

```
1 wsadmin>from javax.swing import JFrame
2 wsadmin>
3 wsadmin>classInfo( JFrame, meth = 'layout' )
4 javax.swing.JFrame
5 | java.awt.Frame
6 | | java.awt.Window
7 | | | java.awt.Container
8 > > > getLayout, setLayout
9 | | | | java.awt.Component
10 > > > > doLayout
```

11 | | | | | java.lang.Object
12 | | | | | java.awt.image.ImageObserver
13 | | | | | java.awt.MenuContainer
14 | | | | | java.io.Serializable
15 | | | javax.accessibility.Accessible
16 | | java.awt.MenuContainer
17 | javax.swing.WindowConstants
18 | javax.accessibility.Accessible
19 | javax.swing.RootPaneContainer

I wasn't too surprised by the presence of getter and setter methods in the layout (i.e., getLayout and setLayout), but I didn't expect to see the doLayout method in the java.awt.Component class. Looking at the Javadoc for the java.awt.Component class, you'll find the information about the doLayout() method[7] shown in Figure 4-2.

public void doLayout()Causes this container to lay out its components. Most programs should not call this method directly, but should invoke the validate method instead.

Figure 4-2. *java.awt.Component.doLayout() description*

[7]See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html#doLayout%28%

That's kind of neat, but I don't intend to be calling that method in any of my code. Let's stick with the recommendation and avoid using it.

Since you looked for the layout methods, it's wise to take a moment to see if any attributes exist that refer to "layout". Listing 4-11 shows how to specify the attr keyword argument for the classInfo() function. The generated output of the JFrame class hierarchy shows that only one attribute exists in this hierarchy that contains the string "layout". It occurs in the java.awt.Container class.

Listing 4-11. JFrame class hierarchy showing "layout" attributes

```
wsadmin>from javax.swing import JFrame
wsadmin>
wsadmin>classInfo( JFrame, attr = 'layout' ) javax.swing.JFrame
| java.awt.Frame
| | java.awt.Window
| | | java.awt.Container
```

```
* * * layout
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

What does this mean? Well, if you look at the Javadoc for java.awt.Container,[8] you might expect to see a "field"[9] called layout. So why isn't one there? Well, you can thank Jython for that. It automatically recognizes a property for which a getter (i.e., getPropertyName or isPropertyName) and a setter (i.e., setPropertyName) method are defined. This allows you to use the PropertyName as an expression or in an assignment statement. You'll use this feature throughout the examples in this book. Let's take a quick look at the example interactive wsadmin session shown in Listing 4-12.

[8]See http://docs.oracle.com/javase/8/docs/api/java/awt/Container.html. [9]Java terminology for what is called an "attribute" in Jython programs. **Listing 4-12.** Huh? What's goin' on, Lucy?

```
1 wsadmin>from java.awt import FlowLayout
2 wsadmin>from javax.swing import JFrame
3 wsadmin>
4 wsadmin>fLayout = FlowLayout()
5 wsadmin>frame = JFrame( 'Title' )
6 wsadmin>frame.layout
7 java.awt.BorderLayout[hgap=0,vgap=0]
8 wsadmin>frame.layout = fLayout
9 wsadmin>frame.layout
10 java.awt.BorderLayout[hgap=0,vgap=0]
11 wsadmin>frame.getLayout()
12 java.awt.BorderLayout[hgap=0,vgap=0]
13 wsadmin>frame.getContentPane().getLayout() 14
```

java.awt.FlowLayout[hgap=5,vgap=5,align=center] 15
wsadmin>frame.getContentPane().getLayout() == fLayout 16 1
17 wsadmin>

Wait a minute! What's going on here? This demonstrates one situation where you have to be mindful of what Jython is trying to do with the "automatic" attribute when it finds getter and setter methods. Table 4-1 describes each line of the wsadmin interactive session shown in Listing 4-12.

Table 4-1. *What's Goin' on Lucy, Explained*

Lines	Detailed Description/Explanation
1-3	Import the AWT and Swing classes that you will be using.
4	Instantiate a FlowLayout Layout Manager object (fLayout).
5	Instantiate a JFrame object and provide a title.
6-7	Use the frame.layout attribute to display information about the default Layout Manager that was created by the JFrame constructor. **Note:** It is a BorderLayout Manager object.
8	Use the frame.layout attribute to assign the kind of Layout Manager you want the frame to use.
9-10	Use the frame.layout attribute to verify the Layout Manager associated with the frame. **Note:** It is *still* a BorderLayout Manager object.
11-12	Use the frame.layout getter method to determine which Layout Manager is associated with the frame. **Note:** It is *still* a BorderLayout Manager object.
13-14	Use the frame.getContentPane() method to obtain a reference to the frame ContentPane, then use its layout getter method to determine the Layout Manager associated with the Content Pane. **Note:** It is a FlowLayout Manager object.
15-16	Verify that the FlowLayout Manager object being used by the Content Pane is, in fact, the same object that was created in line 4. **Note:** The 1 in line 16 means true. The code you are seeing here is due to a special convenience provided by the JFrame class developers.

The sentence from the Javadoc is shown in Figure 4-3.
As a convenience add and its variants, remove and setLayout have been

overridden to forward to the contentPane as necessary.
Figure 4-3. *JFrame "convenience"*
I'm bringing this to your attention now so that you are less likely to be surprised when you stumble upon it while you are developing and maintaining your Jython Swing scripts.

Buttons! Labels! Action!

You've looked at how to create a button and add it to your application. The next thing that you need to figure out is how to make the button do something when it's clicked. When a button is clicked, something called an "event" occurs. A button click is one of many events that can occur in a Swing application. For the time being, I'm going to focus[10] on this specific kind of event. You will learn about other events as they occur in the book.

Listing 4-13 shows a trivial Java application that prints a message (using the System.out.println() method call) when the button is pressed.
Listing 4-13. ButtonDemo_02.java : Reacting to a Button Press

```
1|import java.awt.event.*;
2|import javax.swing.*;
3|public class ButtonDemo_02 {
4| public static void main( String[] args ) {
5| javax.swing.SwingUtilities.invokeLater( new Runnable() {
6| public void run() {
7| JFrame frame = new JFrame( "ButtonDemo" );
8| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
9| JButton button = new JButton( "Press me" ); 10| button.addActionListener(
new ActionListener() { 11| public void actionPerformed( ActionEvent ae ) { 12|
System.out.println( "button pressed" ); 13| };
14| } );
15| frame.add( button );
16| frame.pack();
17| frame.setVisible( true );
18| }
19| });
20| }
21|}
```

Listing 4-14 shows the first attempt at implementing this application in Jython.

One of the things that Java includes is the concept of anonymous classes. Lines 10–14 of Listing 4-13 show how an anonymous class can be used to define an ActionListener, which contains an actionPerformed() method to be invoked when the buttonpressed event occurs. Jython doesn’t allow anonymous classes. Listing 4-14 shows (in lines 7–9) one way to define a ButtonPressed descendent of the ActionListener class. In line 15, an instance of this class is instantiated and added as an action listener to the button. Many (most?) programmers, especially those new to Java, find the use of anonymous inner classes cumbersome and hard to read and write.

[10]Gaining and losing focus are two other kinds of events that can occur.

Listing 4-14. ButtonDemo_01.py: Reacting to a Button Press

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from java.awt.event import ActionListener
5|from javax.swing import JButton
6|from javax.swing import JFrame
7|class ButtonPressed( ActionListener ) :
8| def actionPerformed( self, e ) :
9| print 'button pressed'
10|class ButtonDemo_01( java.lang.Runnable ) :
11| def run( self ) :
12| frame = JFrame( 'ButtonDemo_01' )
13| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) 14| button =
frame.add( JButton( 'Press me' ) )
15| button.addActionListener( ButtonPressed() )
16| frame.pack()
17| frame.setVisible( 1 )
18|if __name__ in [ '__main__', 'main' ] :
19| EventQueue.invokeLater( ButtonDemo_01() )
20| if 'AdminConfig' in dir() :
21| raw_input( '\nPress <Enter> to terminate the application:\n' ) 22|else :
23| print '\nError: This script should be executed, not imported.\n' 24| which = [
'wsadmin -f', 'jython' ][ 'JYTHON_JAR' in dir( sys ) ] 25| print 'Usage: %s
%s.py' % ( which, __name__ )
26| sys.exit()
```

Another approach is shown in Listing 4-15. This example, instead of defining a separate class, uses multiple inheritance. The class defined in line 8 is a descendent of the java.lang.Runnable and the java.awt. event.ActionListener classes. This means that it includes the run() method from the Runnable class and the actionPerformed() method from the ActionListener class. So all you need to do is identify the current class instance as the ActionListener, as shown in line 13.

Listing 4-15. ButtonDemo_02.py: Using Multiple Inheritance

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from java.awt.event import ActionListener
5|from java.lang import Runnable
6|from javax.swing import JButton
7|from javax.swing import JFrame 8|class ButtonDemo_02( Runnable,
ActionListener ) :
9| def run( self ) :
10| frame = JFrame( 'ButtonDemo_02' )
11| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) 12| button =
frame.add( JButton( 'Press me' ) )
13| button.addActionListener( self )
14| frame.pack()
15| frame.setVisible( 1 )
16| def actionPerformed( self, e ) :
17| print 'button pressed'
18|if __name__ in [ '__main__', 'main' ] :
19| EventQueue.invokeLater( ButtonDemo_02() )
20| if 'AdminConfig' in dir() :
21| raw_input( '\nPress <Enter> to terminate the application:\n' ) 22|else :
23| print '\nError: This script should be executed, not imported.\n' 24| which = [
'wsadmin -f', 'jython' ][ 'JYTHON_JAR' in dir( sys ) ] 25| print 'Usage: %s
%s.py' % ( which, __name__ )
26| sys.exit()
```

Although this is pretty interesting, it’s not the only way that this feat can be accomplished. Next, you’ll take advantage of another Jython feature called “keyword arguments.”

Listing 4-16 shows that you don’t even have to declare the class as a descendent of the ActionListener class in order to use it as one (see line 7). You can simply use the implied setter (in line 12) to identify another class method as the ActionListener event handler. The really neat part of this, at least to me, is that you don’t have to call the method actionPerformed(); you can call it something more intuitive, like buttonPressed().

Listing 4-16. ButtonDemo_03.py: Multiple Inheritance Isn’t Required

```
1|import java
2|import sys
3|from java.awt import EventQueue
4|from java.lang import Runnable
5|from javax.swing import JButton
6|from javax.swing import JFrame
7|class ButtonDemo_03( Runnable ) :
8| def run( self ) :
9| frame = JFrame( 'ButtonDemo_03' )
10| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE ) 11| button =
frame.add( JButton( 'Press me' ) )
12| button.actionPerformed = self.buttonPressed
13| frame.pack()
14| frame.setVisible( 1 )
15| def buttonPressed( self, e ) :
16| print 'button pressed'
17|if __name__ in [ '__main__', 'main' ] :
18| EventQueue.invokeLater( ButtonDemo_03() )
19| if 'AdminConfig' in dir() :
20| raw_input( '\nPress <Enter> to terminate the application:\n' ) 21|else :
22| print '\nError: This script should be executed, not imported.\n' 23| which = [
'wsadmin -f', 'jython' ][ 'JYTHON_JAR' in dir( sys ) ] 24| print 'Usage: %s
%s.py' % ( which, __name__ )
25| sys.exit()
```

The biggest difference between using the automatic, or implied, setter call in an assignment statement (e.g., line 12 in Listing 4-16) and using a call to the addActionListener() method (e.g., line 13 in Listing 4-15) is that using the call to the addActionListener() method is necessary when you need to register multiple ActionListener objects.

While you’re looking at some Jython techniques for creating a button, adding it to the frame, and adding an ActionListener, let’s take a look at some other things that you can do using Jython.
Listing 4-17 contains the run() method from two versions of the ButtonDemo_03 script. Lines 1–7 are from the ButtonDemo_03.py script file and lines 9-21 are from the ButtonDemo_03a.py script file.

Listing 4-17. Another Way to Create a Frame and Add a Button

```
1| def run( self ) :
2| frame = JFrame( 'ButtonDemo_03' )
3| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
4| button = frame.add( JButton( 'Press me' ) )
5| button.actionPerformed = self.buttonPressed
6| frame.pack()
7| frame.setVisible( 1 )
8|...
9| def run( self ) :
10| frame = JFrame(
11| 'ButtonDemo_03a',
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 13| )
14| frame.add(
15| JButton(
16| 'Press me',
17| actionPerformed = self.buttonPressed 18| )
19| )
20| frame.pack()
21| frame.setVisible( 1 )
```

In lines 2 and 3 (of Listing 4-17), you can see the technique that has been used up to this point to: Create a JFrame instance.•
• Provide a title string.
• Set the default close operation to be used when the close icon is selected.

This code corresponds with the commonly used Java style. Lines 10–13, on the other hand, show that you can do these same things with a single Jython statement. Granted, since I don’t want the lines to be too long, this statement is being displayed across multiple lines. However, it is still a single statement.
Lines 4 and 5 (again of Listing 4-17) show how you can:

• Create a JButton instance.
• Provide a text string to be displayed on the button.
• Add the button to the frame and save a reference to the button in a variable.
• Define the actionPerformed() method to be called when a button press event occurs.

The code in lines 14–19 does many of these same steps in a single statement. Again, multiple lines are used to make the various parts of the statement easier to read. The only significant thing that this statement doesn't do is save a reference to the button in a variable. This is simply because I didn't see the need to save that information. If you do need to save this value, you can have the code save the result of calling the frame.add() method in a variable.

Updating the Application

It seems kind of silly to have a GUI application display a message to the command prompt window when a button is pressed. This section explains what you need to do in order for the application to make changes to the user display when the button is pressed.

• Add a label to the application, the text of which can be modified by the event handler (the ActionListener actionPerformed()) method.
• Have the actionPerformed (i.e., buttonPressed()) method change the text in the label.

That sounds pretty straightforward, but there's something tricky you need to know about. In order for the buttonPressed() method to access the application label, it needs to be able to identify it and refer to it. So when you create the label, you need to save a reference to it in an instance attribute. Listing 4-18 contains the run() and buttonPressed() methods from the ButtonDemo_04.py application that does these things.

Listing 4-18. Parts of ButtonDemo_04.py

```
11| def run( self ) :
12| frame = JFrame( 'ButtonDemo_04' )
13| frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE )
14| button = frame.add( JButton( 'Press me' ) )
15| button.actionPerformed = self.buttonPressed
```

```
16| self.label = JLabel( 'button press pending' )
17| frame.add( self.label, BorderLayout.SOUTH )
18| frame.pack()
19| frame.setVisible( 1 )
20| def buttonPressed( self, e ) :
21| self.label.setText( 'button pressed' )
```

Table 4-2 explains, in detail, each of the statements in Listing 4-19. The most important ones are in line 16, where a reference is saved to the label instance, and in line 21, where this reference is used to update the label text field.

Table 4-2. *ButtonDemo_04.py Explained*
Lines Detailed Description/Explanation

12-13 These are the same steps used previously to create a JFrame instance and to define the action to take when the close icon is selected.
14-15 These are the same steps used previously to create a button with some text and to identify the method to call when the button press event occurs.
16 Create a JLabel instance and save the reference as an object attribute value (i.e., self.label). 17 Add the JLabel instance to the frame, using the Layout Manager constant (i.e., BorderLayout.SOUTH),[11] to identify where on the application the label should be placed.
18-19 These are the same steps used previously to adjust the application size and to make the application visible.

20-21 ActionListener actionPerformed method that is invoked when a button pressed event occurs. Note how the object instance variable (self.label) is used to modify the text being displayed in the label field.

What happens when you execute this application? Figure 4-4 shows the application, before (on left) and after (on right) the e. This demonstrates that the technique performs as expected.

Figure 4-4. *ButtonDemo_04.py output images.*

Summary

This chapter demonstrates that you can choose to write Jython scripts using syntax that closely matches Java Swing constructs or you can use Jython idioms to make the applications more readable. This means that you should be able to use sample Java Swing applications or code snippets from the Internet to learn how to write and test your Jython Swing applications.

Additionally, you have started looking at some of the many Swing components and learning about how the Swing class hierarchy handles events such as button clicks. Chapter 5 delves into the topic of Layout Managers in great detail; they give you more control over how things are positioned on your application frame.

[11]http://docs.oracle.com/javase/8/docs/api/java/awt/BorderLayout.html. **Chapter 5**

Picking a Layout Manager Can Be a Pane

Earlier in this book,[1] you learned a bit about how items are placed, or positioned, on an application. This chapter deals with that topic in more detail and discusses some of the most commonly used Layout Managers. These include some of the more complicated ones, like the Absolute Layout and GridBagLayout Managers, as well as some of the simpler ones like the FlowLayout, BorderLayout, and BoxLayout Managers.

You will also learn about components that aren't exactly Layout Managers, including SplitPane and TabbedPane. You will then be better able to decide how to position the components on the application window in order to best communicate and interact with your users.

The Absolute Layout Manager Does Not Corrupt Absolutely

Designing and creating graphical applications can be frustrating. This is especially true when you have to position every component on the frame that is displayed to the users. Let's take a look at what this entails.

First, you need to remove any Layout Manager that's already on the container. Then, for each child component, you need to define its size and location on the

frame content pane. Finally, you need to call the container repaint() method to render the components within the container using the current details known about the objects.

This isn't particularly difficult, but it can be tedious and requires a lot of code. Listing 5-1 shows a portion of one way to do this.
Listing 5-1. AbsoluteLayout Example

```
6|class AbsoluteLayout( java.lang.Runnable ) :
7| def run( self ) :
8| frame = JFrame(
9| 'AbsoluteLayout',
10| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 11| )
12| frame.setLayout( None )
13| data = [
14| [ 'A', 20, 10, 0, 0 ],
15| [ 'B', 40, 40, 10, 10 ],
16| [ 'C', 80, 20, 20, 20 ]
17| ]
18| insets = frame.getInsets()
19| for item in data :
20| button = frame.add( JButton( item[ 0 ] ) ) 21| size = button.getPreferredSize()
22| button.setBounds(
23| insets.left + item[ 1 ],
24| insets.top + item[ 2 ],
25| size.width + item[ 3 ],
26| size.height + item[ 4 ]
27| )
28| frame.setSize(
29| 300 + insets.left + insets.right, # frame width 30| 150 + insets.top +
insets.bottom # frame height 31| )
32| frame.setVisible( 1 )
```

[1]Chapter 1 touched on the subject when you added a second label to the frame, In Chapter 4, you saw the layout-related methods and attributes.

How does it work? Well, line 12 shows how to remove any existing Layout Manager. Lines 19-27 show one relatively simple way to create some buttons and define their location and size on the containing frame. It's important to

remember, though, that the call to frame.add() is, in fact, equivalent to calling frame.getContentPane().add(), as discussed in Chapter 3.

When a component (e.g., a button) is created, its constructor will determine its "preferred" size. This preferred size can be obtained by calling the getPreferredSize() method for the component, as you can see in line 21. Then the component's setBounds() method can be used to locate and possibly resize the component.

The first two parameters of the setBounds() method define the component's position by identifying its top-left point. The next two parameters of the method identify the width and height of the component.
What happens when you run this script? Well, you will see three buttons positioned on the application frame. Figure 5-1 shows how the application frame should look. It also shows how challenging it can be to "do it yourself." Did you anticipate that some of the buttons would be overlaid on the screen? I certainly didn't.

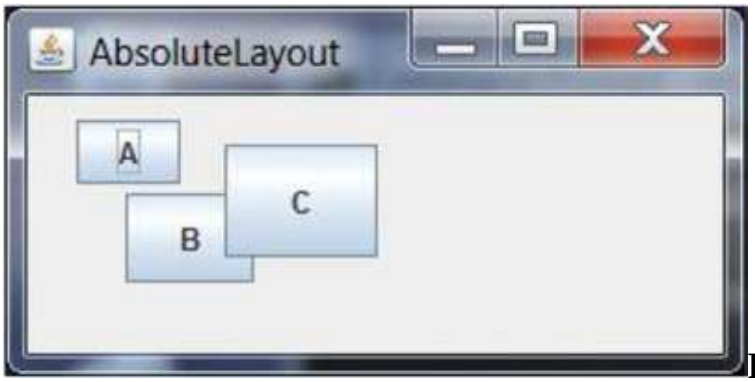


Figure 5-1. *AbsoluteLayout example output*

Are there any advantages to using this technique? Yes, there are. If you resize the application, you will notice that the component's location and size don't change. This is not the case with every Layout Manager, as you will see.

Going with the Flow: The FlowLayout Manager

The next Layout Manager that you're going to investigate is quite simple to use and is called the FlowLayout Layout Manager.[2] This manager attempts to place the components in a row. If there isn't sufficient space in the container to do so, multiple rows will be used. If more than enough space exists, the Layout Manager will, by default, center the components horizontally in the available

space.

Listing 5-2 shows part of the FlowLayoutDemo.py sample application, which demonstrates how this works in an application.

Listing 5-2. FlowLayout Example

```
9|def run( self ) :
10| frame = JFrame(
11| 'FlowLayout',
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 13| )
14| cp = frame.getContentPane()
15| #----------------------------------------------------16| # The alignment can be one of the following values: 17| #-------------------------------------------------------
-18|# cp.setLayout( FlowLayout( FlowLayout.LEFT ) )
19|# cp.setLayout( FlowLayout( FlowLayout.RIGHT ) ) 20| cp.setLayout( FlowLayout() ) # FlowLayout.CENTER = default 21|# cp.setLayout( FlowLayout( FlowLayout.LEADING ) ) 22|# cp.setLayout( FlowLayout( FlowLayout.TRAILING ) ) 23| for name in '1,two,Now is the time...'.split( ',' ) :
24| frame.add( JButton( name ) )
25| #----------------------------------------------------26| # The ComponentOrientation can be either LEFT_TO_RIGHT, or 27| # RIGHT_TO_LEFT. The default is based upon system locale 28| #-----------------
-----------------------------------29|# cp.setComponentOrientation( ... )
30| frame.setSize( 350, 100 )
31| frame.setVisible( 1 )
```

Here are some things to note about this code include:

• The FlowLayout class is part of the java.awt hierarchy.

• A default FlowLayout instance centers the components, with a small space between each.

• The component orientation (left-to-right vs. right-to-left) is inherited from the java.awt.Component class.

Figure 5-2 contains some sample outputs generated using this script.

[2] See http://docs.oracle.com/javase/8/docs/api/java/awt/FlowLayout.html.

Using default values.
Align = RIGHT, Orientation = Right to Left.
Align = RIGHT, Orientation = Right to Left
Too narrow to display the buttons on a single row.

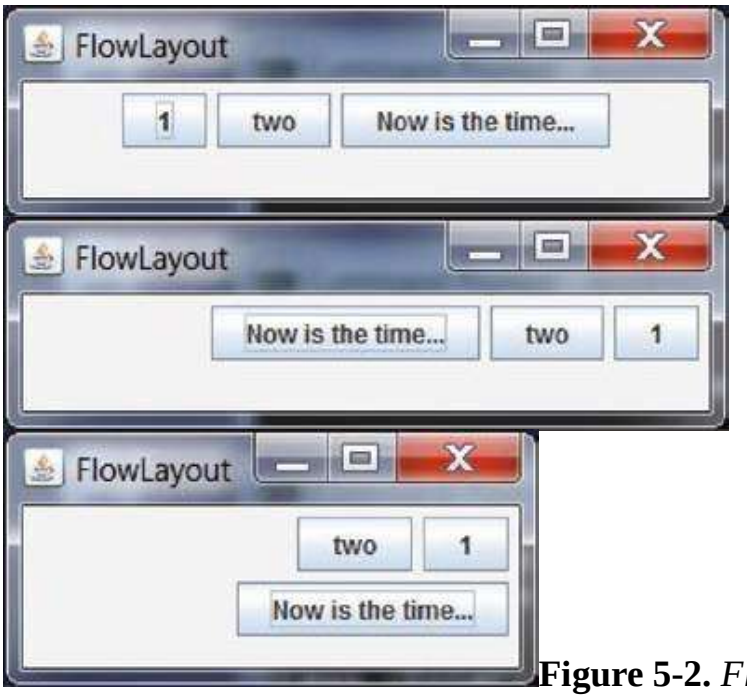


Figure 5-2. *FlowLayout examples*

The description to the left of each image identifies the alignment and orientation values used to produce it. The FlowLayout Layout Manager is much easier than having to specify exactly where components should be placed. However, you must be mindful of what things will look like should the application change size.

South of the Border: The BorderLayout Manager

Next, take a look at the BorderLayout Layout Manager. What do layouts using this Layout Manager look like? Figure 5-3 shows a simple application using this manager.

Figure

5-3. *BorderLayout examples*

The BorderLayout Layout Manager enables you to easily position the components using some simple constants. The labels for each of the buttons shown in Figure 5-3 contain the names of these BorderLayout constants. Additionally, Listing 5-3 shows a portion of the script that generated this output.
Listing 5-3. BorderLayout Example

```
9| def run( self ) :
10| frame = JFrame(
11| 'BorderLayout',
12| layout = BorderLayout(),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 14| )
15| data = [
16| [ 'PAGE_START', BorderLayout.PAGE_START ], 17| [ 'PAGE_END' ,
BorderLayout.PAGE_END ], 18| [ 'LINE_START', BorderLayout.LINE_START
], 19| [ 'LINE_END' , BorderLayout.LINE_END ], 20| ]
21| for name, pos in data :
22| frame.add( JButton( name ), pos )
23| big = JButton(
24| 'CENTER',
25| preferredSize = Dimension( 256, 128 ) 26| )
27| frame.add( big, BorderLayout.CENTER )
28| frame.pack()
29| frame.setVisible( 1 )
```

Tip i encourage you to execute the script and resize the application frame to see what happens to the buttons. remember that the results that you are seeing are because you're using the BorderLayout Layout Manager.

The BorderLayout Layout Manager has advantages, (e.g., simplicity of use) and disadvantages (e.g., the results of resizing the application may not be to your liking). Take these issues into account when you choose your application Layout Manager.

In addition to the constants shown on lines 16-27 in Listing 5-3, BorderLayout also includes some common directional constants. Figure 5-4 shows where these constants position components.

Figure 5-4. *BorderLayout directional constants*

Listing 5-4 shows a portion of the BorderLayoutNEWS.py script that was used to generate this output.

Listing 5-4. BorderLayoutNEWS Example

```
9| def run( self ) :
10| frame = JFrame(
11| 'BorderLayoutNEWS',
12| layout = BorderLayout(),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 14| )
15| data = [
16| BorderLayout.NORTH,
17| BorderLayout.SOUTH,
18| BorderLayout.EAST,
19| BorderLayout.WEST
20| ]
21| for pos in data :
22| frame.add( JButton( pos ), pos )
23| big = JButton(
24| 'Center',
25| preferredSize = Dimension( 256, 128 ) 26| )
27| frame.add( big, BorderLayout.CENTER )
28| frame.pack()
29| frame.setVisible( 1 )
```

The only other point worth noting about the BorderLayout class is that you can

have the Layout Manager automatically separate components by specifying horizontal and vertical gap values when the Layout Manager is instantiated. You can see what this means by looking at Figure 5-5.

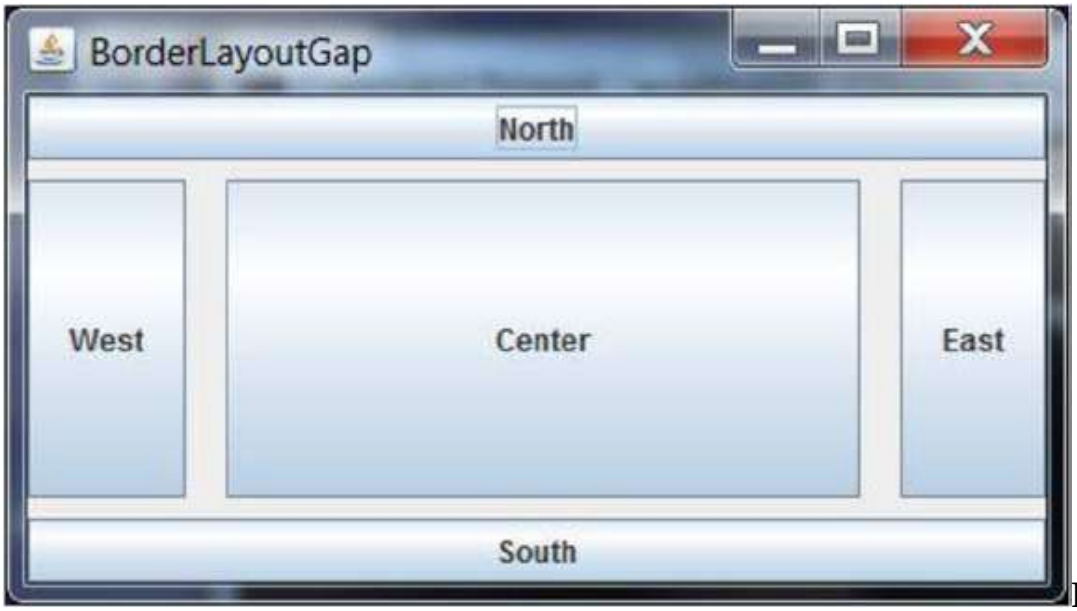


Figure 5-5. *BorderLayout with component separation*

What change was required? All you have to do is specify the gap values on the call to the BorderLayout constructor. The important line from the BorderLayoutGap.py script is shown in Listing 5-5. **Listing 5-5.** BorderLayoutGap Example 12| layout = BorderLayout(16, 8), The constructor used in Listing 5-5[3] identifies the horizontal and vertical gaps to be used between components.

What's in the Cards? Using the CardLayout Manager

The next Layout Manager you'll learn about is the CardLayout Manager. As with the previously described Layout Managers, this one lets you position the components. The interesting thing about this one is that like a deck of cards in a stack, only the top one is visible. What does this mean? Figure 5-6 shows a simple application using the CardLayout Manager. It uses the top row of buttons to allow users to select one of the cards to be displayed.

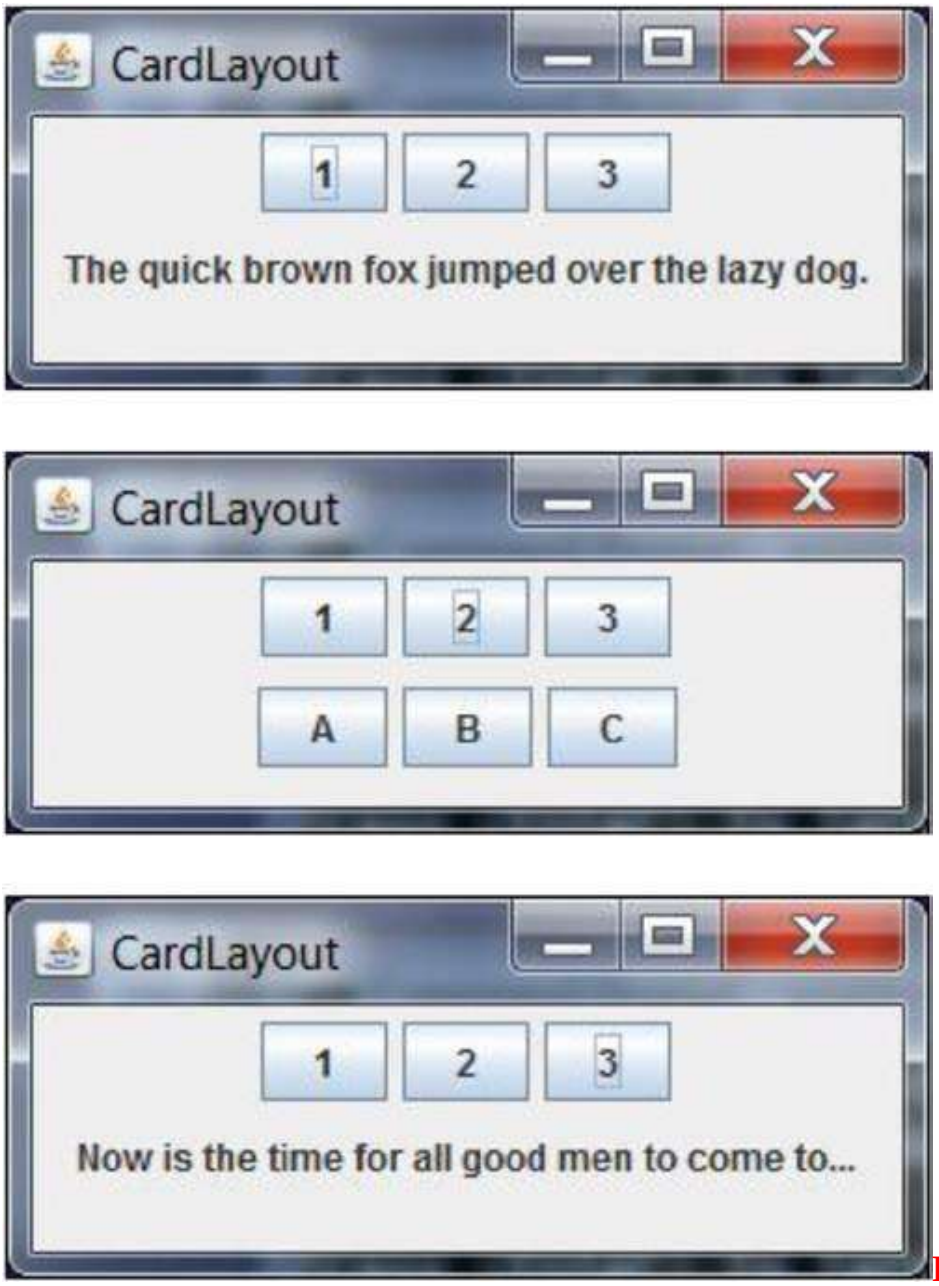


Figure 5-6. *CardLayout examples*

[3]See http://docs.oracle.com/javase/8/docs/api/java/awt/BorderLayout.html#BorderLay

Before you learn about the CardLayout Manager, I first have to quickly mention another container, specifically a JPanel[4] that is used in a number of places in this CardLayout application.

What is a JPanel? It is best to think of a JPanel as a simple, lightweight container

that can be used to hold a group of components. In this example, you will simply create JPanel instances and add some components to each.
Alright, let’s dig into the example. If you click on any of the buttons in the top row, you’ll see that the top panel doesn’t change but the bottom one displays the other components.
How do you do this? Note that the application, (i.e., the JFrame instance) uses one Layout Manager (on its ContentPane). In this case, the application will have two separate parts, or panes. Each pane will be a JPanel instance. The top is used to display the fixed buttons and the bottom is used to display the current CardLayout view.
Listing 5-6 shows the run() method for this example. In it, you can see how:

• The frame layout is configured to use a BorderLayout Manager instance.
• A reference to the frame ContentPane is obtained and the two support routines are called to populate the different parts of the frame panel.
Listing 5-6. CardLayoutDemo run() Method

```
10|class CardLayoutDemo( java.lang.Runnable ) :
11| def run( self ) :
12| frame = JFrame(
13| 'CardLayout',
14| layout = BorderLayout(),
15| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
16| )
17| cp = frame.getContentPane()
18| self.addButtons( cp, BorderLayout.NORTH )
19| self.addCards( cp, BorderLayout.CENTER )
20| frame.setSize( 300, 125 )
21| frame.setVisible( 1 )
```

Listing 5-7 shows how the addButtons() method uses the default JPanel Layout Manager, i.e., a FlowLayout Manager instance, to add three buttons to the panel. The choice of button labels will become clear shortly. Please note that the ActionListener actionPerformed method for each button is assigned to be the buttonPress() method in this object instance. It is important to realize that each container component in the application has its own, possibly default, Layout Manager. This allows the application designers to position the individual components as they see fit.

Listing 5-7. CardLayoutDemo addButtons() Method

```
22| def addButtons( self, container, position ) :
23| panel = JPanel()
24| for name in '1,2,3'.split( ',' ) :
25| panel.add(
26| JButton(
27| name,
28| actionPerformed = self.buttonPress
29| )
30| )
31| container.add( panel, position )
```

Finally, the addButtons() method adds the button panel to the user-specified container, i.e., the frame ContentPane, using the specified location or position. [4]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JPanel.html.

Listing 5-8 shows how the addCards() method uses a default JPanel instance for each of the card panels. For simplicity's sake, the first and last of these card panels contain only a label, which can be used to verify which card panel is being displayed. The middle card panel contains three more buttons to show how easily a card panel can be populated with components.

Listing 5-8. CardLayoutDemo addCards() Method

```
32| def addCards( self, container, position ) :
33| card1 = JPanel()
34| card1.add(
35| JLabel(
36| 'The quick brown fox jumped over the lazy dog.'
37| )
38| )
39| card2 = JPanel()
40| for name in 'A,B,C'.split( ',' ) :
41| card2.add( JButton( name ) )
42| card3 = JPanel()
43| card3.add(
44| JLabel(
45| 'Now is the time for all good men to come to...'
```

```
46| )
47| )
48| cards = self.cards = JPanel( CardLayout() )
49| cards.add( card1, '1' )
50| cards.add( card2, '2' )
51| cards.add( card3, '3' )
52| container.add( cards, position )
```

Finally, another JPanel is created to hold the various card panels that you just created and populated. Each of these card panels is added to the panel that is using the CardLayout Layout Manager. Note that when a panel is added, you specify an identifier to be used to select this panel. These strings correspond to the button labels used in the addButtons() method shown in Listing 5-7.

Listing 5-9 shows how the buttonPress() event-handling method can obtain a reference to the Layout Manager object instance being used by the panel. Then this Layout Manager's show() method can be used to specify a card panel instance to be displayed. The label of the button that was pressed can be used by calling the event.getActionCommand() method. All in all, this makes for a very clear, concise, and precise event handler.

Listing 5-9. CardLayoutDemo buttonPress() Method

```
53| def buttonPress( self, event ) :
54| deck = self.cards.getLayout()
55| deck.show( self.cards, event.getActionCommand() )
```

Hopefully you can see how easily this can be used to create a really neat application. Next, you will take a look at some of the various components that your applications can use.

Splitting Up Is Easy to Do: Using Split Panes

Unlike splitting the atom, Swing provides for a very easy way to partition all of, or part of, your application into two pieces. Unfortunately, it's not a Layout Manager, but it certainly applies to the topic at hand, so I cover it here anyway. The JPanel that was mentioned earlier provides you with a lightweight container for keeping a collection of

components together. Wouldn’t it be neat if you had a simple way of splitting a panel in two, either vertically or horizontally? Well, there is a way, called a JSplitPane.[5] Figure 5-7 shows a very simple application with two buttons, one in each of the two parts of the horizontally split panel.

Figure 5-7. *JSplitPane sample output*

One feature that might not be immediately obvious in Figure 5-7 is that the separator (the divider) between the two parts of the split pane is moveable. The image on the right shows how the application appears after moving the separator as far to the right as possible. The limitation is imposed by the minimum size of each button.

Listing 5-10 shows just how easy it is to create and use one of these split panes in your applications.

Listing 5-10. SplitPane1: Simple Horizontal Separation

```
7|class SplitPane1( java.lang.Runnable ) :
8| def run( self ) :
9| frame = JFrame(
10| 'SplitPane1',
11| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
12| )
13| frame.add( JSplitPane(
14| JSplitPane.HORIZONTAL_SPLIT,
15| JButton( 'Left' ),
16| JButton( 'Right' )
17| )
18| )
19| frame.pack()
20| frame.setVisible( 1 )
```

Vertical Splits: Not as Painful as They Sound

It is just as easy to have a portion of your application window separated vertically. Look at line 14 in Listing 5-10. You can see how the JSplitPane class

contains a constant that indicates in which direction (horizontally or vertically) the pane should be split. If, instead of using JSplitPane.HORIZONTAL_SPLIT, you use JSplitPane.VERTICAL_SPLIT, your application will look similar to Figure 5-8.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSplitPane.html.

Figure 5-8. *JSplitPane2, vertical split*

Unlike in the previous application, this separator doesn't move. Well, that's only partially true. If you resize the window, as shown in Figure 5-9, you can see how the top part remains the same and all of the additional height is given to the bottom part. Then, you can move the separator bar down, resizing the two parts of the split panel. Again, each portion of the split pane is limited by the minimum size restriction of the components.

Figure 5-9. *JSplitPane2, vertical split and*

resizing
Limited Resources: Setting Size Attributes

A long time ago, in a galaxy far, far away, I took an economics class that used something called the “Guns versus butter model”[6] to describe the concept of choosing between two limited resources. In the same way, split panes restrict the amount of space that the separator bar can move. How is this determined? Every Swing component has the JComponent[7] and the Component[8] as some of its base classes. Therefore, each Swing component has different size attributes, as you can see in Listing 5-11. These attributes are used to identify things like the minimum, maximum, and preferred size of a component.

Listing 5-11. Component size attributes

```
wsadmin>from javax.swing import JButton
wsadmin>
wsadmin>classInfo( JButton, attr = 'size' )
javax.swing.JButton
| javax.swing.AbstractButton
| | javax.swing.JComponent
| | | java.awt.Container
| | | | java.awt.Component
* * * * ancestorResized, baselineResizeBehavior * * * * componentResized,
maximumSize, maximumSizeSet * * * * minimumSize, minimumSizeSet,
preferredSize * * * * preferredSizeSet, size
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver
| | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | java.io.Serializable
| | java.awt.ItemSelectable
| | javax.swing.SwingConstants
| javax.accessibility.Accessible
wsadmin>
```

What are all of those “size” attributes in the java.awt.Component class, anyway? Well, the size attribute identifies the current width and height of the component, which shouldn’t be too much of a surprise. The others—minimumSize, maximumSize, and preferredSize—are suggestions, or hints,[9] to the Layout

Manager about how the particular component should be represented.

The effect of the minimumSize attribute can be the borne out with a little testing. Figure 5-10 shows the initial application image, including the separators between the split panes. Each button in the application uses the same event handler routine to display the button text, the current button size, and the other size attributes mentioned previously. I encourage you to use the application to improve your understanding about the various component size attributes.

[6] See http://en.wikipedia.org/wiki/Guns_versus_butter_model.
[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html.
[8]See http://docs.oracle.com/javase/8/docs/api/java/awt/Component.html.
[9]"…the code is more what you'd call 'guidelines' than actual rules." From http://www.imdb.com/title/tt0325980/quotes.

Figure 5-10. *SplitPane4: initial rendering*

To use the application, you need only click on the buttons to display the button size attributes. Then you can resize the window and use the buttons again to see which attribute values have changed. When the window size increases, you can also move the split pane divider lines by dragging them. Again, you can use the buttons to see the effect on the button size attributes.

Nested Split Panes
The application that illustrates the divider movement seen in Figure 5-10 uses the simple concept of nested split panes. Listing 5-12 shows the part of the SplitPane4.py script that produced this output.
Listing 5-12. Nested Split Panes

The button() method, on lines 17, 20, and 21, is a local reference to a method used to create a JButton with the actionPerformed() method assigned. The actionPerformed() method is the event handler used to display the button component sizes.

One interesting thing from this listing is how the top component (i.e., the second argument to the JSplitPane constructor) in line 17 is a simple call to the JButton constructor. Notice how the next argument, (i.e., the third for the JSplitPane constructor), which is used to specify the bottom component for the vertically split pane, is itself a JSplitPane constructor. This is the nesting to which this

section refers.

Imagine how much fun you could have using this technique to create deeply nested split pane components![10] If you search the web, you can find an interesting article, written by Hans Muller entitled “MultiSplitPane: Splitting Without Nesting.”[11] If you are really adventurous, you might investigate adding his class files to your environment and using them or converting them to Jython. If you do, please let me know how it goes.

Divider and Conquer

What, exactly, can you do with, and about, the separator (aka divider) bar? As you saw earlier, the Layout Manager decides how to allocate the available space to display the components. Do you have any control over the divider bar, and how it can be used? Sure, there are a number of JSplitPane methods related to the divider.

For example, Figure 5-11 shows how you can easily adjust the size of the divider bar using the setDividerSize() method. The image corresponds to the selected size button (i.e., the top image has a divider size of zero, the middle image has a divider size of 10, and the bottom image has a divider size of 20).

[10]This was discussed in Chapter 3.

[11]See http://today.java.net/pub/a/today/2006/03/23/multi-split-pane.html. **Figure 5-11.** *SplitPane5: various divider sizes*

One SplitPane attribute with which you may be unfamiliar is the OneTouchExpandable property. By default, this Boolean attribute is set to false. When it is true, the divider has two little triangles added to it, as shown in Figure 5-12.

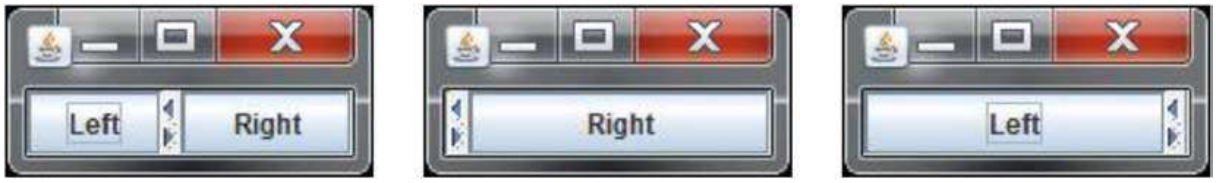


Figure 5-12. *SplitPane6 with the* ***OneTouchExpandable*** *divider*

By enabling OneTouchExpandable, you allow the user to minimize one component with a single mouse click, which is really quite cool. Looking at the script used to produce this output, which can be found in the SplitPane6.py script file, you can see that you only needed to specify the oneTouchExpandable

keyword argument on the JSplitPane constructor call. Listing 5-13 shows how easily this can be accomplished using Jython.

Listing 5-13. oneTouchExpandable Keyword Argument

```
15| frame.add(
16| JSplitPane(
17| JSplitPane.HORIZONTAL_SPLIT,
18| JButton( 'Left' ),
19| JButton( 'Right' ),
20| oneTouchExpandable = 1
21| )
22| )
```

Rules for Using Split Panes

Unlike a during knife fight, there are rules for using a split pane. Most of them deal with how to determine the component sizes, and therefore the divider bar. This can also lead to dealing with the sizes of any nested components, which can be a challenge. In fact, there is a wonderful statement on the Java tutorials page entitled “How to Use Split Panes”[12] that says:

■ **Note** Choosing which sizes you should set is an art that requires understanding how a split pane’s preferred size and divider location are determined.

Immediately after this statement, the page contains a bulleted list of rules for making split panes work well for you and your applications. If you intend to use split panes, I suggest that you bookmark that URL and study it often. I certainly have.

Can I Run a Tab? Using a TabbedPane

This section discusses something else that really isn’t a Layout Manager. It seems to be related to the CardLayout Layout Manager you read about earlier. In fact, it’s a different kind of panel. The reason I am discussing it here is because of the similarities it has to the CardLayout Manager mentioned in the previous section. Figure 5-13 shows how JTabbedPane can be used to display one of a group of panes based on the user’s selection. Unlike the CardLayout Layout Manager, an instance of JTabbedPane has a selectable tab label that can be used to determine the content to be displayed.

[12]Seehttp://docs.oracle.com/javase/tutorial/uiswing/components/splitpane.html.

Figure 5-13. *TabbedPaneDemo examples*

To demonstrate the similarities and differences between a panel using the CardLayout Manager and a TabbedPane class, the code shown in Listing 5-14 uses the same child panel contents as those seen in Listing 5-8.

Listing 5-14. TabbedPaneDemo Class

```
10|class TabbedPaneDemo( java.lang.Runnable ) :
11| def run( self ) :
```

```
12| frame = JFrame(
13| 'TabbedPaneDemo',
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| self.addTabs( frame.getContentPane() )
17| frame.setSize( 300, 125 )
18| frame.setVisible( 1 )
19| def addTabs( self, container ) :
20| tab1 = JPanel()
21| tab1.add(
22| JLabel(
23| 'The quick brown fox jumped over the lazy dog.' 24| )
25| )
26| tab2 = JPanel()
27| for name in 'A,B,C'.split( ',' ) :
28| tab2.add( JButton( name ) )
29| tab3 = JPanel()
30| tab3.add(
31| JLabel(
32| 'Now is the time for all good men to come to...' 33| )
34| )
35| tabs = JTabbedPane()
36| tabs.addTab( 'Uno' , tab1 )
37| tabs.addTab( 'Dos' , tab2 )
38| tabs.addTab( 'Tres', tab3 )
39| container.add( tabs )
```

What's the main difference? Well, in lines 36-38, you can see how the child panes are added to the JTabbedPane instance using the addTab() method instead of the JPanel add() method shown in lines 49-51 of Listing 5-8.
A JTabbedPane instance is also easier because the selection mechanism is built into the class. Take another look at Listings 5-7 and 5-9, where you had to specifically identify how the CardLayout panel instances would be selected by the users.

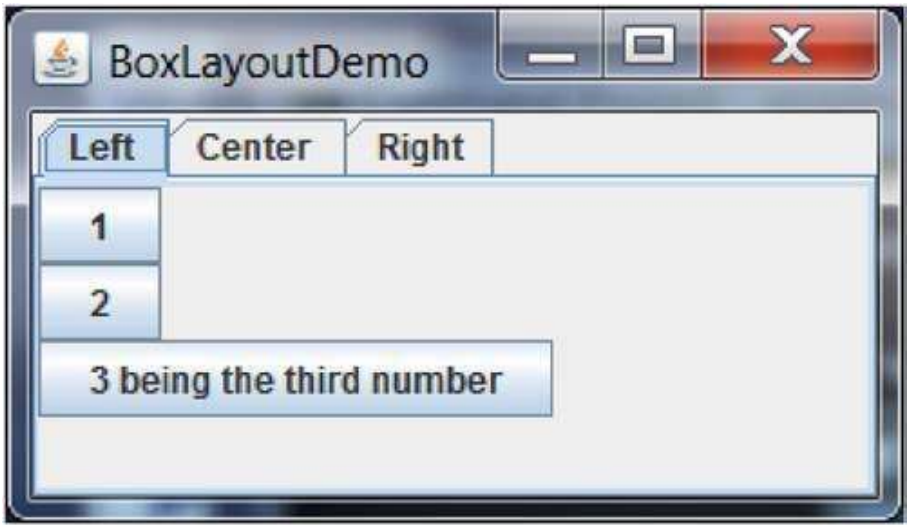
Are You Boxed In? Using the BoxLayout Manager

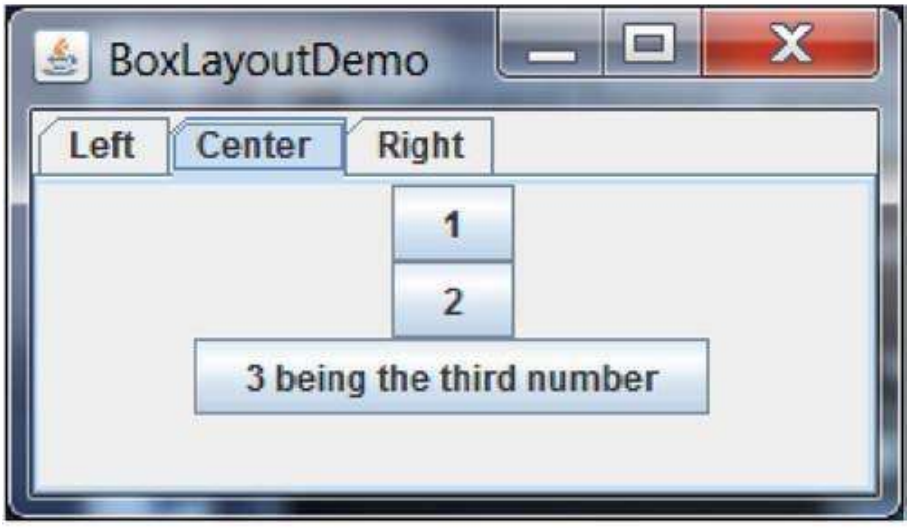
Earlier in this chapter, you saw how easy it was to use the FlowLayout Layout Manager to position the components in a row. You also saw that when the

application window was resized, the Layout Manager would reposition the components on different rows when the width of the application window would otherwise be too narrow. There are times when you don’t want this to occur. This leads to the next Layout Manager, i.e., the one defined in the BoxLayout class.[13]

Figure 5-14 shows the sample output of the BoxLayoutDemo sample application. It uses a JTabbedPane with three tabs. In each tab, you can see three buttons that are layered out along the Y_AXIS (i.e., vertically). The only difference between the tabs is how the buttons are aligned within the pane.

[13]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/BoxLayout.html.


BoxLayoutDemo
Left
Center
Right
1
2
3 being the third number


BoxLayoutDemo
Left
Center
Right
1
2
3 being the third number

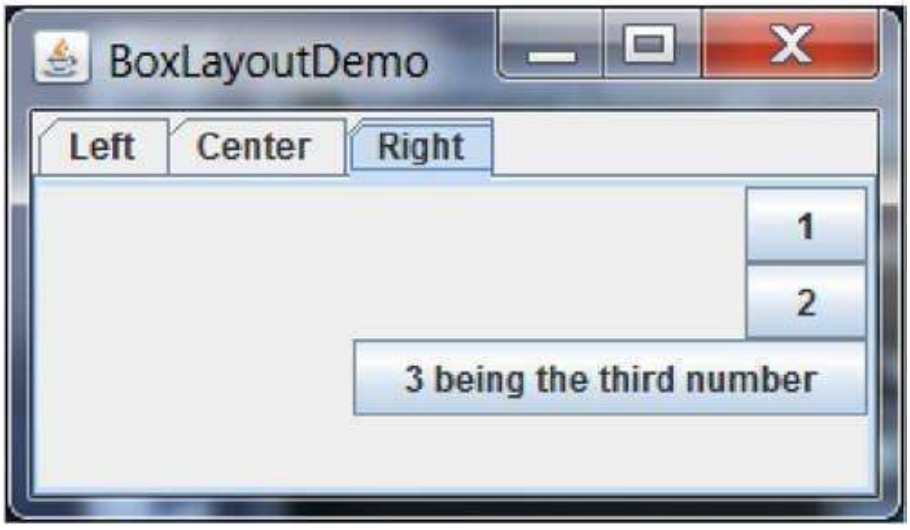

BoxLayoutDemo
Left
Center
Right
1
2
3 being the third number

Figure 5-14. *BoxLayout tabs*

Listing 5-15 shows the BoxLayoutDemo class that produces the output shown in Figure 5-14. As you can see, when you instantiate the Layout Manager, as shown in line 29, you tell it how the components should be displayed. Later, in line 31, you see that when you instantiate the buttons, you can use a Component alignment constant to tell the component where the other part should be displayed (i.e., along the x-axis of the panel).

Listing 5-15. BoxLayoutDemo Class

```
10|class BoxLayoutDemo( java.lang.Runnable ) :
11| def run( self ) :
12| frame = JFrame(
13| 'BoxLayoutDemo',
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| self.addTabs( frame.getContentPane() )
17| frame.setSize( 300, 175 )
18| frame.setVisible( 1 )
19| def addTabs( self, container ) :
20| align = [
21| [ 'Left' , Component.LEFT_ALIGNMENT ],
22| [ 'Center', Component.CENTER_ALIGNMENT ],
23| [ 'Right' , Component.RIGHT_ALIGNMENT ]
24| ]
25| names = '1,2,3 being the third number'.split( ',' )
26| tabs = JTabbedPane()
27| for aName, aConst in align :
28| tab = JPanel()
29| tab.setLayout( BoxLayout( tab, BoxLayout.Y_AXIS ) )
30| for name in names :
31| tab.add( JButton( name, alignmentX = aConst ) )
32| tabs.addTab( aName, tab )
33| container.add( tabs )
```

The constants provided by the BoxLayout class include those listed in Table 5-1.

Table 5-1. *BoxLayout Constants* **Constant**

BoxLayout.X_AXIS BoxLayout.Y_AXIS BoxLayout.LINE_AXIS

BoxLayout.PAGE_AXIS

Description
Components are laid out horizontally, left to right.
Components are laid out vertically, top to bottom.

Components are laid out based on the container's ComponentOrientation property, either horizontally or vertically.
Components are laid out the way text is displayed on a page, based on the container's ComponentOrientation property, either horizontally or vertically.
The Box Class

Even though this chapter has been focusing mainly on Layout Managers, you have also learned about some container classes to help define how the application components are displayed. The Box class is another of these lightweight containers that helps arrange application components.

If you need to create a single row or column of components, you might want to take a look at the Box class,[14] which works kind of like a JPanel. One significant difference between these two classes is that the Box class can *only* use a BoxLayout Layout Manager,[15] whereas the JPanel allows the developer to choose any kind of Layout Manager.[16]

Building a Box

The Box class has a single constructor, as shown in Figure 5-15. One of the interesting things about this constructor is that it isn't used as often as the createHorizontalBox() and createVerticalBox() methods. I think that calling the Box.createHorizontalBox() method is more understandable than Box(BoxLayout.X_AXIS), don't you? Can you think of a reason that you might prefer the constructor to the createHorizontalBox() method?

public Box(int axis)
Creates a Box that displays its components along the specified axis.
Parameters:
axis - can be BoxLayout.X_AXIS, BoxLayout.Y_AXIS, BoxLayout.LINE_AXIS or BoxLayout.PAGE_AXIS.
Throws:
AWTError -if the axis is invalid

See Also:
createHorizontalBox(), createVerticalBox()
Figure 5-15. *Box constructor*

If nothing comes to mind, you might want to take another look at Table 5-1. The important thing to note is that the createHorizontalBox() class is equivalent to using the Box(BoxLayout.X_AXIS) constructor call, which will position the components left to right, regardless of the user locale. This might not be what you intend. You have to be the judge.

[14] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/Box.html.
[15]See
http://docs.oracle.com/javase/8/docs/api/javax/swing/Box.html#setLayout%28jav
[16]By default, a JPanel instance will use a FlowLayout Layout Manager.

Invisible Box Components

One of the challenges with using a Box instance to hold your components is the fact that they are, by default, adjacent to one another. Figure 5-16 shows what happens when three fixed-size components (i.e., JLabel instances) are added to a horizontal box container. The left image shows how the application looks when the application is first made visible. The right image shows what happens to the components when the application is widened.

Figure 5-16. *Horizontal (fixed-size) components*
If you were expecting the components to be centered or right-aligned, you aren't going to be pleased with the results. To help adjust this kind of alignment, the Box class includes the following "filler" components:

- Glue
- Struts
- Rigid areas

Your choice of filler component will depend upon the initial appearance of your box, as well as on how you want it to look when it is resized.
Glue

If you research Java Swing glue components, most of the references mention the fact that glue is not a very good description for this type of component. This is true because most people think of glue as a kind of adhesive. Instances of this invisible Swing component will start out with a width or height of zero. This attribute then increases in size as the container size increases.

Three Box methods exist that can be used to create glue components. These methods are listed and described in Table 5-2.

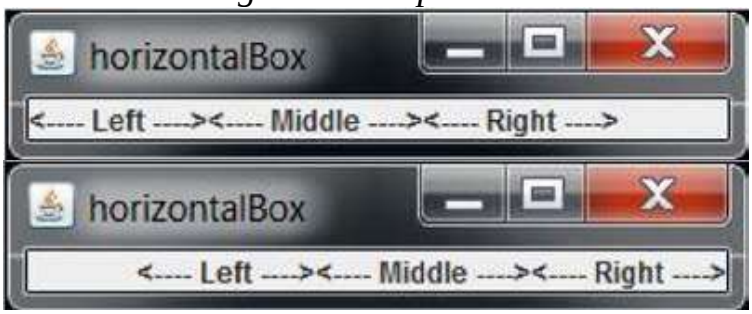
Table 5-2. *Box Class Glue Methods* **Method Name** createGlue()

createHorizontalGlue() createVerticalGlue()
Description/Details
Used to create a component that can be used between horizontal or vertical components.
Used to create a component that is oriented horizontally.
Used to create a component that is oriented vertically.

Glue components have a minimum size (horizontal width or vertical height) of zero. This means that it shouldn't be obvious when a glue component is between other components on the box container. When the container becomes larger, the Layout Manager will increase the size of all of the resizable components (i.e., the ones smaller than their maximum size). When multiple fixed-size components are positioned between glue components, the available space will be allocated equally to the glue components.

It may help to see an example to better understand what this means. Table 5-3 shows what happens when glue components are positioned around some fixed-size components (i.e., JLabel instances), and the application frame containing the horizontal box is widened.

Table 5-3. *Using Glue Components in a Horizontal Box*



No glue components exist, so all the “extra” space is allocated to the right of the last label.
One glue component is added before the first label, so it gets all the “extra” space.

Glue components are added before and after the first label. The “extra” space is equally distributed between the glue components.

Glue components are added in three places. The “extra” space is equally distributed among these glue components.

Glue components are added before, after, and in between the labels. All the “extra” space is allocated equally.
When a vertical box is used to hold components, glue components act in a similar fashion.
Strut Components
There are times when it is appropriate to have some minimum distance between components. For those times, consider using horizontal or vertical struts. Table 5-4 shows two methods that you can use to create these strut components.

Table 5-4. *Box Class Strut Methods*
Method Name Description/Details
createHorizontalStrut(int width) Create a filler component of the specified width (in pixels). createVerticalStrut(int height) Create a filler component of the specified height (in pixels).

Is there any problem related to using strut components? Well, I have found some references that indicate that horizontal strut components have an unlimited height, and vertical strut components have an unlimited width, which can be a

problem. Apparently, if you use strut components with nested vertical and horizontal box containers, alignment issues can arise. Take a look at this example to see what they are talking about. Figure 5-17 shows an application that has five buttons (all of which have the same text) aligned in a plus pattern. The issue isn't obvious until you drag the bottom-right corner of the application down to increase the size and width. As you can see from the image on the right, the results are unexpected. The glue components are placed before the leftmost button, after the rightmost one, above the top one, and beneath the bottom one. When the frame becomes larger, the strut component before the rightmost button has extra space.

Figure 5-17. *Alignment issues with strut components*

Rigid Area Components

The final invisible component—called a rigid area—is created using specific, fixed dimensions. This means that the component size should not change even if its container is resized. This would seem to make it the preferred filler component if your application is best rendered with some space between visible components. Table 5-5 shows the Box method that you can use to create this type of invisible component.

Table 5-5. *Rigid Area Creation Method*

Method Name	Description/Details
createRigidArea(Dimension d)	Creates an invisible component that's always the specified size.

Boxes and Resizable Components

Up to this point, you have been using fixed-size components when dealing with

box containers (e.g., labels and buttons). It is important to note, however, that some components have a maximum size that isn't the same as the minimum and preferred size. What happens when that occurs?

Based on what I have already discussed, you probably realize that the Layout Manager will also allocate available space to these components as space becomes available. The distribution of the available size depends on a number of factors, including:

• The maximum size attribute of the resizable components
• The current size of the container
• The size of the screen

I don't know about you, but I was a little surprised about this last one. In fact, I had to try it out, so I played around with the fBox.py script, which displays information about the text field width when the button is pressed. When the maximum width of the text field is less than or near the screen width, the amount of space allocated to the text field as the application width increases is relatively small. As the maximum width of the text field is increased, you'll see that the amount of space allocated to the text field as the application is widened increases accordingly. If the maximum width of the text field is not limited (i.e., if the default maximum width is used), the text field will receive all of the available extra width.

Gridlock, Anyone? Using the GridLayout Manager

Another common layout configuration is when the components are laid out in a grid. The GridLayout Layout Manager provides this configuration. Figure 5-18 shows a simple example where the application has two panes, each of which contains a group of buttons displayed in a grid. Each grid has three columns and enough rows to display all of the buttons.


GridLayoutDemo
Horizontal: 0
Horizontal: 2
Horizontal: 4
Horizontal: 8
Horizontal: 16
Vertical: 0
Vertical: 2
Vertical: 4
Vertical: 8
Vertical: 16


GridLayoutDemo
Horizontal: 0
Horizontal: 2
Horizontal: 4
Horizontal: 8
Horizontal: 16
Vertical: 0
Vertical: 2
Vertical: 4
Vertical: 8
Vertical: 16


GridLayoutDemo
Horizontal: 0
Horizontal: 2
Horizontal: 4
Horizontal: 8
Horizontal: 16
Vertical: 0
Vertical: 2
Vertical: 4
Vertical: 8
Vertical: 16

Figure 5-18. *GridLayoutDemo examples*

The interesting thing about this application is that the buttons allow you to dynamically change the horizontal or vertical gap between the buttons on each pane that is using the GridLayout Layout Manager.
Listing 5-16 shows the run() method in the GridLayoutDemo application. In this method, you can see how the main panel uses a BoxLayout Layout Manager to position two inner panes along the Y_AXIS (i.e., vertically). Then, the addButtons() method is called to add another inner pane, which uses a GridLayout Layout Manager to position a group of buttons.

Listing 5-16. GridLayoutDemo run() Method

```
def run( self ) :
frame = JFrame(

'GridLayoutDemo',
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
main = JPanel()
main.setLayout( BoxLayout( main, BoxLayout.Y_AXIS ) )
self.panes = []
self.addButtons( main, 'Horizontal:' )
self.addButtons( main, 'Vertical:' )
frame.add( main )
frame.setSize( 500, 250 )
frame.setVisible( 1 )
```

Listing 5-17 shows how the addButtons() method creates a new panel, populates it with buttons, and then adds it to the specified container. A reference to the new panel is saved in the self.panes list, created in the run() method, for use by the button event handler.

Listing 5-17. GridLayoutDemo addButtons() Method

```
def addButtons( self, container, prefix ) :
pane = JPanel( GridLayout( 0, 3 ) )
self.panes.append( pane )
for size in '0,2,4,8,16'.split( ',' ) :

pane.add(
```

```
JButton(
'%s %s' % ( prefix, size ),
actionPerformed = self.buttonPress

)
)
container.add( pane )
```

Listing 5-18 shows how the button label is retrieved using the event.getActionCommand() method. The program then uses the space that exists between the direction (i.e., Horizontal or Vertical) and the size (i.e., the number of pixels) for the inner component gap.

Listing 5-18. GridLayoutDemo buttonPress() Method

```
def buttonPress( self, event ) :
dir, size = event.getActionCommand().split( ' ' ) if dir[ 0 ] == 'H' :

for pane in self.panes :
layout = pane.getLayout()
layout.setHgap( int( size ) )
layout.layoutContainer( pane )

else :
for pane in self.panes :
layout = pane.getLayout() layout.setVgap( int( size ) ) layout.layoutContainer(
pane )
```

This is where you need to process each of the panes using a GridLayout Layout Manager. Given a pane, you can obtain a reference to the Layout Manager used by that pane using the pane.getLayout() method call. You can use this reference to adjust the horizontal or vertical gap used by this Layout Manager (i.e., by calling the setHgap() or setVgap() method).

Once the gap has been changed, a call is made to the layoutContainer() method to force the Layout Manager to adjust how the components are displayed.

Shaking Things Up: The GridBagLayout Manager

One of the problems with the GridLayout Manager is that sometimes you'll want

there to be some differences between the size and placement of the components. For example, you might want things to be arrayed in rows, but not use the regular column arrangement provided by GridLayout. You can use the GridBagLayout Layout Manager for these purposes. Unfortunately, it comes with a price. Specifically, in order for you to have more control over the size and placement of components in a grid, you must provide a greater level of detail to each component. Therefore, the GridBagLayout Manager is a bit more complicated than the others.

Figure 5-19 shows sample output of the GridBagLayout applications found in the code\Chap_05 directory. These are just examples of the kind of control that the GridBagLayout Layout Manager provides for component placement.

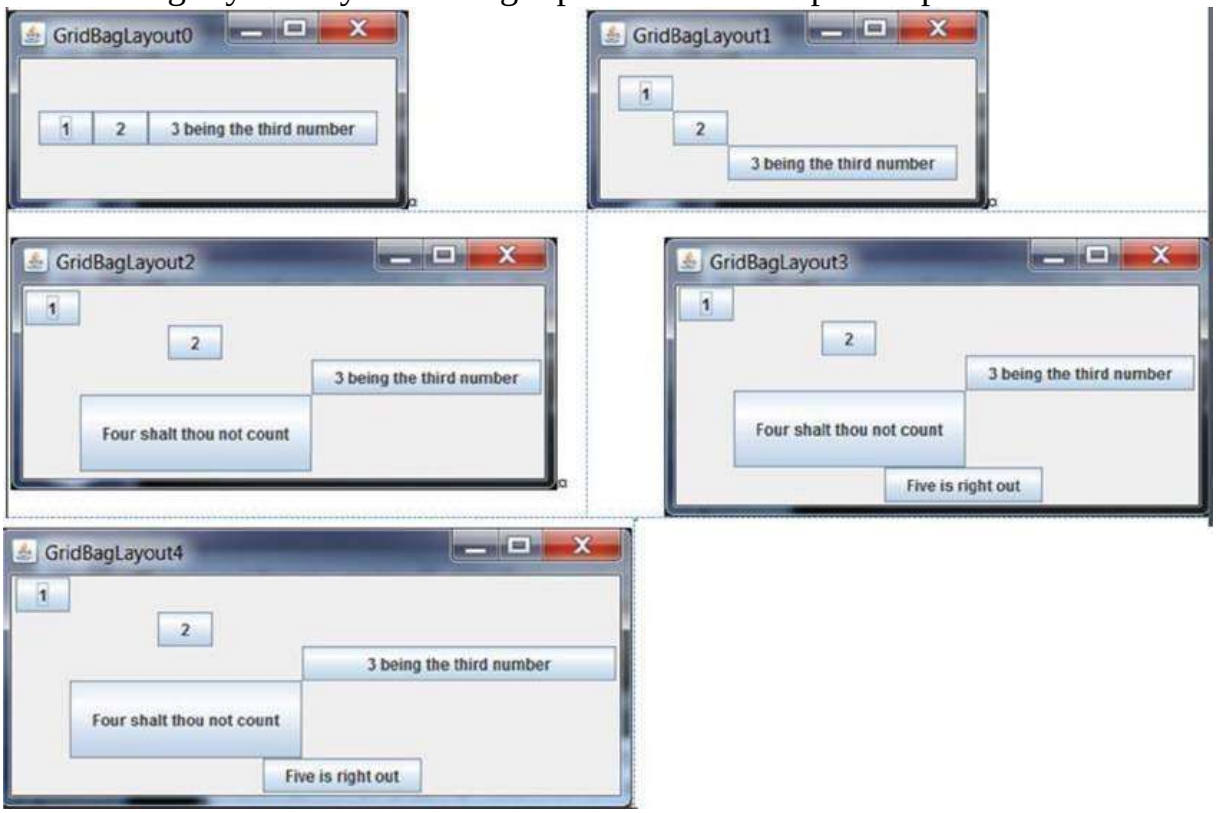


Figure 5-19. *GridBagLayout examples*

Listing 5-19 shows the addComponents() method from the GridBagLayout4 script, which was used to generate the output shown in Figure 5-19. There are some important points about this method to note. For example, a new GridBagConstraints instance is created for each component. Why is this considered a “best” practice? This is recommended because the reuse of existing GridBagConstraints objects can easily lead to subtle, and therefore difficult-to-

diagnose, problems. It is all too easy to forget that one of the many constraint fields has a non-default value.

Listing 5-19: GridBagLayout4 addComponents() Method

```
def addComponents( self, container ) :
container.setLayout( GridBagLayout() )
c = GridBagConstraints()
c.gridx = 0 # first column
c.gridy = 0 # first row
container.add( JButton( '1' ), c )
c = GridBagConstraints()
c.gridx = 1 # second column
c.gridy = 1 # second row
container.add( JButton( '2' ), c )
c = GridBagConstraints()
c.fill = GridBagConstraints.HORIZONTAL
c.gridx = 2 # third column
c.gridy = 2 # third row
c.weightx = 0.0
c.gridwidth = 3
container.add( JButton( '3 being the third number' ), c ) c = GridBagConstraints()
c.gridx = 1 # second column
c.gridy = 3 # forth row
c.ipady = 32 # make this one taller
container.add( JButton( 'Four shalt thou not count' ), c ) c =
GridBagConstraints()
c.gridx = 1 # second column
c.gridy = 4 # fifth row
c.gridwidth = 3 # make this one 3 columns wide container.add( JButton( 'Five is
right out' ), c )
```

This provides me with another opportunity to highlight some of the features and strengths of Jython. If you take a look at the Javadoc for the GridBagConstraints class,[17] you will see that two constructors exist. The first constructor has no parameters and creates a GridBagConstraints instance using all of the default values. The other has 11 parameters and requires that all of the values be provided.

Jython, on the other hand, lets you use keyword arguments to selectively provide only those values of interest. Listing 5-20 shows an interactive session that uses a displayConstraints() function[18] to show any non-default constraint values that exist in the specified object. In lines 8, 14, and 19, you can see how keyword arguments can be provided when the object is instantiated.

[17]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/GridBagConstraints.html.
[18]The complete source for which can be found in code\Chap_05\displayConstraints.py script file. **Listing 5-20.** Specifying GridBagConstraints parameters using keywords

```
wsadmin>from java.awt import GridBagConstraints
wsadmin>from java.awt import Insets
wsadmin>
wsadmin>c = GridBagConstraints()
wsadmin>displayConstraints( c )
All constraint values match defaults
wsadmin>
wsadmin>c = GridBagConstraints( gridx = 1, gridy = 2 )
wsadmin>displayConstraints( c )
Non-default constraint values:

gridx: 1
gridy: 2
wsadmin>
wsadmin>c = GridBagConstraints( insets = Insets( 1, 2, 3, 4 ) )
wsadmin>displayConstraints( c )
Non-default constraint values:
insets: java.awt.Insets[top=1,left=2,bottom=3,right=4] wsadmin>
wsadmin>c = GridBagConstraints( fill = GridBagConstraints.CENTER )
wsadmin>displayConstraints( c )
Non-default constraint values:
fill: CENTER
wsadmin>
```

Looking at Other Layout Managers

In addition to the Layout Managers you already read about, there is the

SpringLayout class[19]. I'm not going to cover it here because of its complexity, as well as because it's most often used by GUI builders. At this time, I am unaware of any GUI builder that generates Jython code, so it would seem that your best bet is to find one that generates Java code and translate it to the corresponding Jython code. Unfortunately, I don't know how many people would invest the time to do this, especially if the application's "look and feel" changes frequently.

Another Layout Manager, called GroupLayout,[20] exists, and is also used by GUI builders. It too is beyond the scope of this book due to its complexity. If you are interested in learning more about it, read the section in the Java Swing tutorials called "How to Use GroupLayout."[21]

The other alternative that isn't covered here is the "roll your own" option. There is good information about this approach at your fingertips; see the section in the Java Swing tutorials entitled "Creating a Custom Layout Manager."[22] This, too, is beyond the scope of this book, and I don't have enough experience with it to feel comfortable trying to tackle this topic here.

[19] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/SpringLayout.html.
[20]See http://docs.oracle.com/javase/8/docs/api/javax/swing/GroupLayout.html.
[21]See http://docs.oracle.com/javase/tutorial/uiswing/layout/group.html.
[22]See http://docs.oracle.com/javase/tutorial/uiswing/layout/custom.html.

Summary

This chapter is all about providing you with information to help you understand how you can use Layout Managers or containers to position your application components. As you can see, there are a number of options open to you and your applications. It's all about how you want the application components to be arranged. One of the aspects discussed in each section is how the components change when the container is resized. This is an important topic to consider when you are designing and developing your application, so keep that in mind.

Chapter 6

Using Text Input Fields

Up to this point, you've been limited in the kinds of applications that you can

build because of the lack of ways of getting information (i.e., input) into the application. You will now start to remedy that deficiency by learning to create an input field that lets the users enter text.

This chapter discusses a couple of input-related fields such as JTextField and JTextArea. Additionally, you will take a look at how the data is displayed e.g., using alignment and fonts. The chapter also covers creating a simple Swing application that allows you to display and modify a value in the WebSphere Application Server (WSAS) environment. One reason to do this is so that you can see how to perform “long-running” operations on a separate thread using instances of the SwingWorker class.

What Does It Take to Get Data Into an Application?

To start, you’ll create a fairly simple value that you want to be able to view and possibly modify. How about the WebSphere Administrative Console inactivity timeout? Well, unfortunately, the WebSphere documentation only contains one sample Jacl script that shows how this can be done.[1] You’ll start by creating two Jython functions, one to get the current inactivity timeout value and the other to set it.

Listing 6-1 shows an interactive wsadmin session where these functions are used to get and set the timeout value. The complete functions can be found in code\Chap_06 directory.[2] They’ve been tested, as shown in Listing 6-1, so you should be all set to use them, right?

Listing 6-1. Getting and setting the admin console inactivity timeout

```
wsadmin>print getTimeout()
30
wsadmin>print setTimeout( '123' ) Successfully modified.
wsadmin>print getTimeout()
123
wsadmin>print setTimeout( '30' ) Successfully modified.
wsadmin>
```

[1] See http://www14.software.ibm.com/webapp/wsbroker/redirect?version=compass&product=was-nddist&topic=cons_sessionto.

[2] In files named getTimeout.py and setTimeout.py, respectively.

JTextField: Getting Data Into the Application

To make things easy, you will start with a simple text field that solicits information from the users. What is required and how does it work? You start by creating a JTextField instance, which provides a simple, lightweight input field that allows users to enter a small amount of text. When the user input is complete, which is generally indicated by the user pressing the Enter key, any associated ActionListener will receive an action event.[3]

If you look at the Java documentation for the JTextField class,[4] you'll see that a number of constructors exist. The simplest of these requires no parameters. One of those enables you to provide an initial string value to be displayed, one allows the field width in number of columns, and another allows the initial string value to be specified as well as the number of columns to be displayed. This application will use something simple—the one that allows you to specify the number of columns to be displayed. Since you are unlikely to need more than three digits, you can use a value of three for the number of columns.

To provide the users with visual indications of what the input field contains, you need to surround it with some labels. And to be a little more complete, you can have a message label field on a separate line to display a message indicating the success or failure of the requested action.

■**Note** do you remember how you were able to use the actionPerformed keyword assignment as part of the constructor call when you created a JButton? if not, have a look at Chapter 4.[5]
You can use the same technique to specify the ActionListener routine to be called when an actionEvent occurs. In the case of JTextField, this is when the user presses the Enter key.

Your First, Almost Real, Application

What does it take to use one or more existing routines and turn them into a graphical wsadmin application? Well, after a bit of work, you're likely to take what you have learned, including the information about the JTextField, and build an application, the output of which might look something like Figure 6-1.

Figure 6-1. *consoleTimeout1 sample output*

[3] I haven't covered action events in great detail, at least not yet. They are covered in Chapter 14. However, you have used the actionPerformed() method when you learned about buttons earlier.

[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextField.html.

[5]For example, take a look at Listing 4-8.

What code do you need to do this? Listing 6-2 shows the consoleTimeout1 class from the sample application script file. Note how it calls the getTimeout() and setTimeout() routines, just like the interactive session shown in Listing 6-1. This makes it look pretty simple, doesn't it?

Listing 6-2. consoleTimeout1 Class

```
class consoleTimeout1( java.lang.Runnable ) : def run( self ) :
frame = JFrame(
'Console timeout',
defaultCloseOperation = JFrame.EXIT_ON_CLOSE )
cp = frame.getContentPane()
cp.setLayout( BoxLayout( cp, BoxLayout.Y_AXIS ) )

input = JPanel( layout = FlowLayout() )
input.add( JLabel( 'Timeout:' ) )
self.text = JTextField( 3, actionPerformed = self.update ) input.add( self.text )
self.text.setText( getTimeout() )
input.add( JLabel( 'minutes' ) )
cp.add( input )

self.msg = cp.add( JLabel() )
frame.setSize( 290, 100 ) frame.setVisible( 1 )

def update( self, event ) :
value = self.text.getText().strip()
if re.search( '^\d+$', value ) :
```

```
self.msg.setText( setTimeout( value ) ) else :
msg = 'Invalid value "%s" ignored.' % value
self.msg.setText( msg )
self.text.setText( getTimeout() )
```

Unfortunately, it isn't as simple as it looks. If you test the application and watch closely, you are likely to notice a delay between the time that you type a value and press Enter, and the message being displayed. What's happening?
Welcome to the world of event-driven applications. What you are seeing is the fact that a non-trivial delay is occurring between the time that the setTimeout() routine is being called and when it returns. In the meantime, the entire application is hung. This is a really bad thing, and exhibits a terrible practice.

Help Me SwingWorker, You're My Only Hope!

A first attempt at fixing this might involve changing the message field to show some kind of message. You might even want to disable the input field before calling the setTimeout() routine. So, how might you do this? Well, you could replace the call to the setTimeout() routine in Listing 6-1 with something like what's shown in Listing 6-3. **Listing 6-3.** Attempt to Fix the update() Method

```
self.msg.setText( 'working...' ) self.text.setEnabled( 0 )
self.msg.setText( setTimeout( value ) ) self.text.setEnabled( 1 )
```

What do the calls to the setEnabled() method associated with the JTextField do? The first one attempts to disable it by specifying 0, which is interpreted as false, and the second call attempts to enable it by specifying 1, which is interpreted as true.[6]

If it worked, this would disable the input field so that the AdminConfig scripting object could locate and modify the inactivity timeout value associated with the appropriate configuration object.
Unfortunately, this doesn't work because the part of the Swing framework that updates the screen never has a chance to gain control to update the display. How do you fix this? How do you force the application to perform some separate action while the Swing framework displays the GUI and allows the user to generate events like button clicks?
Well, to do this, I need to take a trip down the rabbit hole and talk about something called *concurrency*.[7] It's similar to when you have multiple separate

programs running on your computer at the same time. In this case, you need the application to have multiple things going on. The challenge is that all of this needs to be happening within one program, the wsadmin Jython script.
When multiple programs execute at the same time, the operating system and the program developer are responsible for keeping things straight, so that two or more programs don't try to manipulate the same piece of information (e.g., an object or variable) at the same, or nearly the same time.
How do you have separate things going on at the same time in your Jython scripts? Well, you are going to have to make use of a concurrent programming concept called *threads*. You need to identify operations that may require a "long" time to complete and have these executed on a separate thread. A *thread* is a short form of, or nickname for, a thread of control. Each thread operates separately and distinctly.
Since I don't have the time or space to completely cover this topic, I am going to try to provide just enough so that you can resolve the common issues that you'll likely encounter when using the applications discussed.
The really good news is that Jython scripts are executing on a Java Virtual Machine, and Java was created with concurrency in mind. This means that your scripts can make use of classes and techniques that have been part of Java for years.
The primary class that you need to use is called SwingWorker. The Javadoc for the SwingWorker class[8] includes a simple case example that translates to Listing 6-4.

[6] Just like you do with the call to frame.setVisible(1) in the application run() method.
[7]Concurrent programming is a large topic all by itself. I don't have the time or space to cover it here completely. You will, however, learn enough to be able to develop graphical Jython applications. There are lots of good articles available on the topic. I encourage you to search for "threads and concurrent programming."
[8]See http://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html.

Listing 6-4. Simple SwingWorker Subclass

```
class InTheSwing( SwingWorker ) :
def __init__( self, labelField = None ) : self.label = labelField
def doInBackground( self ) :
try :
```

self.result = longRunningRoutine() except :
self.result = 'Exception encountered.' def done(self) :

self.label.setText(self.result) ...
label = frame.add(JLabel())
InTheSwing(label).excecute()

How well does this work? The images shown in Figure 6-2 can answer this question. You can see that when there's an invalid value entered (in this case "x"), the bad value is replaced with the original value and an appropriate error message is displayed. The next image shows what happens when a valid value is entered. The input field has been disabled and a "working..." message is displayed, at least until the update is complete. The last image shows that the input field has been enabled; the "Update successful" message is displayed.

Figure 6-2. *consoleTimeout2 Sample Output*

What code do you need to do this? Listing 6-5 shows the WSAStask class, which is a SwingWorker descendent class. How does it work?

Listing 6-5. WSAStask Class

```
58|class WSAStask( SwingWorker ) :
59| def __init__( self, textField, labelField ) :
60| self.text = textField # Save the References
61| self.label = labelField
62| SwingWorker __init__( self )
63| def doInBackground( self ) :
64| self.text.setEnabled( 0 ) # Disable input field
65| self.label.setText( 'working...' ) # Inform user of status
66| value = self.text.getText().strip()
```

67| if not re.search(re.compile('^\d+$'), value) : 68| msg = "Invalid numeric value "%s",' % value 69| self.label.setText(msg)
70| self.text.setText(getTimeout())
71| else :
72| self.label.setText(setTimeout(value)) 73| def done(self) :
74| self.text.setEnabled(1) # Enable input field

The explanation of how this class works can be found in Table 6-1. Fortunately, the code that needs to be executed on a separate thread of control is quite easy to understand.

Table 6-1. *WSAStask Class, Explained*
Lines Description/Explanation

59–62 Class constructor used to save references to the necessary component fields and call the SwingWorker (i.e., base class) constructor (i.e., the SwingWorker.__init__() method).

63–72 Method called by the SwingWorker execute() method that does the actual, possibly long, running work. It is interesting to note how calls to global functions (i.e., getTimeout() and setTimeout()) are made here. Note also how the field references saved by the constructor method (i.e., the __init__() method) are used here to access and manipulate the actual components.

73–74 The done() method is also called by the SwingWorker execute() method when the doInBackground() method completes.

What does that do for your consoleTimeout class? In Listing 6-2, the update() method required half a dozen steps. Listing 6-6 shows how you only need to instantiate a WSAStask object and call its execute method to do the work. Remember that all of the stuff that was previously done in the update() method has been moved to the WSAStask doInBackground() method. So the code to perform the work is now performed on the separate thread of control.

Listing 6-6. consoleTimeout2's update() Method def update(self, event) : WSAStask(self.text, self.msg).execute()
Warning it's important to note that SwingWorker objects cannot be reused. additional work, even if it is identical to what was done previously, must be done using a completely new instance of the SwingWorker descendant class.

Back to the JTextField

Up to this point, I haven't provided many details about the JTextField component. The reason for this was to allow you to quickly use a simple kind of input field for the sample application. Now, however, it's appropriate to revisit the JTextField class, so you can garner a more complete understanding of its capabilities, limitations, and uses.

If you look into the Java Swing Tutorial, you'll can find a section entitled, "How to Use Text Fields"[9] Let's start by describing and using the JTextField, which is the simplest of the text input fields.
Looking at the JTextField Javadoc shows that it includes a number of methods. Unfortunately, I don't have the time or space to completely describe each and every method. However, there are some that warrant investigation. One getter/setter pair that is likely to catch your eye is the one dealing with horizontal alignment. Figure 6-3 shows the output of the TextAlignment.py[10] script, which illustrates the various JTextField alignment values and how they affect text display in a JTextField.

Figure 6-3. *TextAlignment output*

Listing 6-7 shows the TextAlignment class from the script used to display this output. It is interesting to note how easily you can display this information using a GridLayout. You don't even have to tell the Layout Manager how many rows will be displayed. By initializing the number of rows as 0, as shown on line 12, you let the Layout Manager keep track of how many rows are provided.

Listing 6-7. TextAlignment Class

```
8|class TextAlignment( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(
```

```
11| 'TextAlignment',
12| layout = GridLayout( 0, 2 ),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
14| )
15| data = [
16| [ 'Left' , JTextField.LEFT ],
17| [ 'Center' , JTextField.CENTER ],
18| [ 'Right' , JTextField.RIGHT ],
19| [ 'Leading' , JTextField.LEADING ],
20| [ 'Trailing', JTextField.TRAILING ]
21| ]
22| for label, align in data :
23| frame.add( JLabel( label ) ) 24| text = frame.add(
25| JTextField(
26| 5,
27| text = str( align ), 28| horizontalAlignment = align 29| )
30| )
31| frame.pack()
32| frame.setVisible( 1 )
```

[9]See http://docs.oracle.com/javase/tutorial/uiswing/components/textfield.html.
[10]The complete script is in the Code\Chap_06\TextAlignment.py file.

The other thing to note here is how you can use the Jython keyword parameter assignments (lines 12, 13, 27, and 28) to greatly simplify the code.
Consider the difference between the Left and Leading alignments. Remember that you might be building applications that are used in a variety of locations across the world. Left and Right are absolute direction indicators, whereas the Leading and Trailing alignments are related to the locale and the direction that text should be displayed, based on the locale setting.

Size Matters: Looking at Text Font Attributes

While you're looking at the JTextField documentation, it's smart to consider also the setFont() method. Can you change the font used by the JTextField components? With a tiny bit of work, I was able to change the TextAlignment.py example to use a different font for each of the JTextField values. The output of this modification is shown in Figure 6-4.

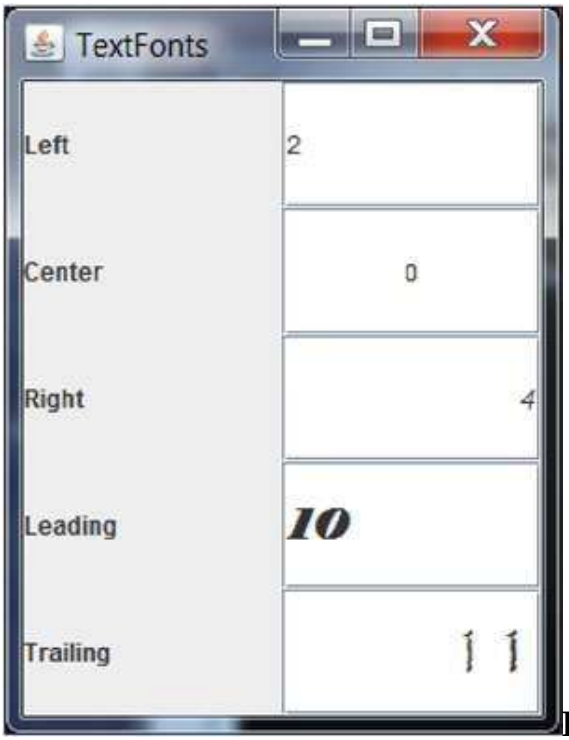


Figure 6-4. *TextFonts output*

The changes you need to make to generate this output are shown in Listing 6-8. This isn't anything nearing a complete discussion about fonts; it's just a glimpse as to how easy they are to change.

Listing 6-8. TextFonts Changes

```
data = [
[ 'Left' , JTextField.LEFT , None ],
[ 'Center' , JTextField.CENTER , Font( 'Courier' , Font.BOLD, 12 ) ], [ 'Right' ,
JTextField.RIGHT , Font( 'Ariel' , Font.ITALIC, 14 ) ], [ 'Leading' ,
JTextField.LEADING , Font( 'Elephant', Font.BOLD | Font.ITALIC, 20 ) ], [
'Trailing', JTextField.TRAILING, Font( 'Papyrus' , Font.PLAIN, 36 ) ]

]
for label, align, font in data :
frame.add( JLabel( label ) )
text = frame.add(
JTextField(
5,
text = str( align ),
horizontalAlignment = align,
)
)
```

```
if font :
text.setFont( font )
```

This code adds an additional column for each row in the data array. The value of this column is None or a font instance. If a font instance is present, it is used to define the font to be used for the associated JTextField instance. The test and associated assignment are performed in the last two lines of Listing 6-6.

The Elephant (Font) in the Room

One of the questions that might come to you as you are looking at this code is, “Is there really a font named ‘Elephant’”? I was a little surprised by that one as well. This made me wonder what it would take to create a simple application that displays the list of available font names.

First, you need some kind of data area that can be used to hold a bunch of information. For this purpose, you’re going to use the next kind of input field, called a JTextArea. One of the things to note about this particular application is that it uses an output field, not an input field.

How do you do that? The Java documentation for this component[11] includes half a dozen constructors, half of which allow you to provide a string to be used to initialize the text. Listing 6-9 shows how you can obtain the list of available fonts from a local graphics environment instance. One thing that you have to note, however, is that you have to convert this list of strings into a single string, with newline characters delimiting each line of text. Fortunately, a simple Jython idiom exists to do exactly this, as you can see in line 19.

[11]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JTextArea.html.

Listing 6-9. AvailableFonts Class

```
8|class AvailableFonts( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(

11| 'Available Fonts',
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
13| )
14| lge = GraphicsEnvironment.getLocalGraphicsEnvironment()
```

```
15| fontNames = lge.getAvailableFontFamilyNames()
16| frame.add(
17| JScrollPane(
18| JTextArea(
19| '\n'.join( fontNames ),
20| editable = 0,
21| rows = 8,
22| columns = 32
23| )
24| )
25| )
26| frame.pack()
27| frame.setVisible( 1 )
```

The other important thing to notice is that this text area is likely to contain more lines of text than will comfortably fit on the application screen. Therefore, you need to put the text area instance in a container that will automatically provide vertical and horizontal scroll bars, as needed, to comfortably display a reasonable application window. You do this using an instance of the JScrollPane class.[12]

Is it always this simple and easy? Well, it all depends. In this case, you can use almost all of the default settings. The other really good thing about Jython is that you can also use keyword arguments to help document what the parameters mean. So, all in all, you benefit from the well-designed Swing classes hierarchy.

Take a moment and comment out lines 21 and 22. Don't forget to remove or comment out the trailing comma on line 20 as well.[13] Before you execute the script, what do you expect it to display? When I ran the modified script, it didn't match my expectations. I don't know about you, but I think that the use of the keyword arguments on the JTextArea constructor call certainly made the application better looking. This just to point out to you that not all of the Swing default values will match your expectations or needs. Try to keep this in mind.

Using JTextArea for Input

Let's talk for a moment about the editable keyword assignment in line 20 of Listing 6-7. By now, you should be able to recognize it as equivalent to a call to the setEditable() method with a value of false. All this so the area of text can't be

modified by the user. All you have to do to make this input component editable is remove this keyword assignment or change the value from 0 (false) to 1 (true).

What happens if you execute the script after making this change? Well, you can select, modify, remove, or add text. And all it really takes is a JTextArea instance. With the JTextField, as you saw earlier in the chapter, you can add an ActionListener in order for an event handler routine of your choice to be invoked when the user presses Enter.

[12] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JScrollPane.html.
[13]You could replace lines 16–25 with the following statement, which I find much easier to read: frame.add(JScrollPane(JTextArea('\n'.join(fontNames), editable = 0)))

Which listeners make sense for a JTextArea? I don't know. Let's use the classInfo() routine, seen earlier, to find out what kind of listeners can be added. Oops. That's one of the problems about asking questions. You may not want to see the answer you get. What are all of these things? Table 6-2 briefly explains the kinds of listeners that can be added.

Table 6-2. *JTextArea Listeners* **Listener Name**
AncestorListener
CaretListener
ComponentListener
ContainerListener
FocusListener
HierarchyBoundsListener HierarchyListener
InputMethodListener
KeyListener
MouseListener

MouseMotionListener MouseWheelListener PropertyChangeListener
VetoableChangeListener

Listener Description
Support notification when changes occur to a JComponent or one of its ancestors. Listens for changes in the caret position of a text component.
The listener interface for receiving component events.
The listener interface for receiving container events.

The listener interface for receiving keyboard focus events on a component. The listener interface for receiving ancestor moved and resized events. The listener interface for receiving hierarchy changed events.
The listener interface for receiving input method events.
The listener interface for receiving keyboard events (keystrokes).

The listener interface for receiving “interesting” mouse events (press, release, click, enter, and exit) on a component.

The listener interface for receiving mouse motion events on a component. The listener interface for receiving mouse wheel events on a component. A PropertyChange event gets fired whenever a bean changes a “bound” property.

A VetoableChange event gets fired whenever a bean changes a “constrained” property.

If you chose to have a listener for each kind of event, you would be able to micromanage the JTextArea instance. I’m not even going to investigate all of these listeners. I am, however, going to take a look at the CaretListener, since it can be used to monitor changes to the current position in the JTextArea portion of your application.

What does that require? Well, looking at the CaretListener Javadoc,[14] you see that only one method, caretUpdate(), needs to be implemented.
You can see that as you type into the input area, the label at the bottom of the application is updated to reflect the number of words and lines that exist. Figure 6-5 shows the sample output after pasting some text into the input area.

[14]See http://docs.oracle.com/javase/8/docs/api/javax/swing/event/CaretListener.html.

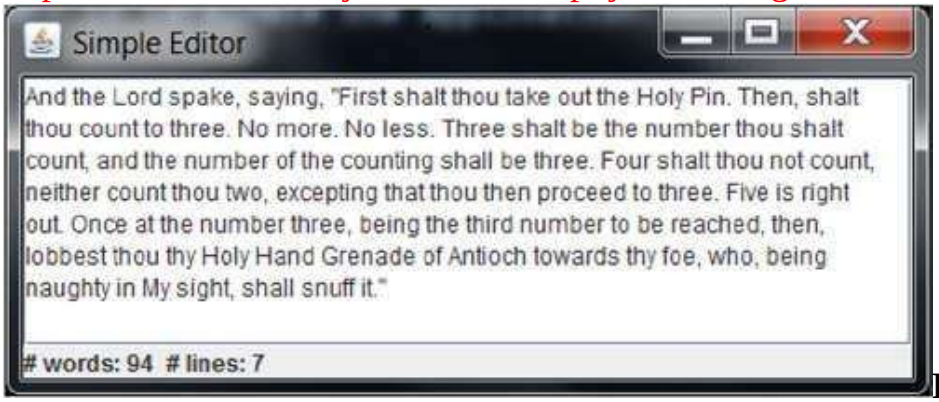


Figure 6-5.

SimpleEditor output

Listing 6-10 shows just how easy it is to use a TextArea and a CaretListener to create a very simple text editor.

Listing 6-10. SimpleEditor Class

```
class SimpleEditor( java.lang.Runnable ) :
def run( self ) :
frame = JFrame(

'Simple Editor',
layout = BorderLayout(),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
self.area = JTextArea(
rows = 8,
columns = 32,
caretUpdate = self.caretUpdate
)
frame.add( JScrollPane( self.area ), BorderLayout.CENTER )
self.words = JLabel( '# words: 0 # lines: 0' )
frame.add( self.words, BorderLayout.SOUTH )
frame.pack()
frame.setVisible( 1 )

def caretUpdate( self, event, regexp = None ) :
if not regexp :
regexp = re.compile( '\W+', re.MULTILINE )

pos = event.getDot()
text = self.area.getText()
if text.strip() == '' :

words = lines = 0
else :
words = len( re.split( regexp, text ) )
lines = len( text.splitlines() )
msg = '# words: %d # lines: %d' % ( words, lines )
self.words.setText( msg )
```

■ **Warning** this caretUpdate() method does not work well with large amounts of data. each time the caret (cursor) moves, the method is invoked, and it retrieves the entire text area and uses a regular expression to separate the data into “words.” the splitlines() method then separates the data into “lines.” this is very inefficient and suitable only for a trivial example such as this one.

Summary

This chapter is the first one to discuss the use of text input fields for your application scripts. It also introduced the important topic of threads, and the use of SwingWorker class instances to perform potentially long-running operations on a separate thread to keep the application from hanging. In the next chapter, you’ll take a look at some other components that allow the users to provide input.

Chapter 7

Other Input Components

This chapter presents other input components available in the Swing class hierarchy. Specifically, it deals with some specialized text input fields, such as JPassword, three types of ComboBoxes (static, editable, and dynamic), and formatted text fields. Additionally, I combine some of these while discussing the JSpinner class at the end of the chapter.

Password Fields

One common reason for having a fairly small input field is to allow the users to enter a password. The good news is that the Swing class hierarchy includes an input component specifically for this purpose. Having recently seen the JTextField component, you shouldn’t be too surprised to learn that a descendent component exists, called JPasswordField.[1] The bad news is that working with passwords, especially within a scripting language such as Jython, includes the possibility of security exposures, especially if the script writer doesn’t think in terms of potential vulnerabilities.

Fortunately, the Swing developers did keep security in mind when designing Java as well as this input component.[2] For example, you may not realize it, but

unlike a normal text input field, the text in a JPasswordField can’t be cut or copied. If you try to do so, a little bell sound is played to indicate that the requested action cannot be performed.

JPasswordField is a descendent of JTextField, which was discussed in Chapter 6. What happens when text is entered into JPasswordField? Figure 7-1 shows that as you enter text, each character is replaced by a userconfigurable echo character. Each instance of this component can, if you choose, have a unique echo character. To identify the character to be displayed for this input field, you use the setEchoChar() method.

Figure 7-1. *PasswordDemo sample output*

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JPasswordField.html.

[2]Unfortunately, there are lots of ways for script writers to overlook good security practices, so be careful.

What does it take to use JPasswordField? Listing 7-1 contains the PasswordDemo class used to generate the output shown in Figure 7-1. There are some things I need to mention about this code though. Notice how, on line 13, the size keyword attribute is used to define the size of the frame to be displayed. This is especially useful in an example like this, which includes an initially empty JLabel component that will be used on the application window to display a message.

Listing 7-1. PasswordDemo Class

```
9|class PasswordDemo( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'PasswordDemo',
13| size = ( 215, 100 ),
14| layout = FlowLayout(),
15| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 16| )
```

```
17| frame.add( JLabel( 'Password:' ) )
18| self.pwd = frame.add(
19| JPasswordField(
20| 10,
21| actionPerformed = self.enter
22| )
23| )
24| self.msg = frame.add( JLabel() )
25| frame.setVisible( 1 )
26| def enter( self, event ) :
27| print 'ActionCommand: "%s"' % event.getActionCommand() 28| pwd =
self.pwd.getPassword()
29| if pwd == jarray.array( 'test', 'c' ) :
30| result = 'correct'
31| else :
32| result = 'wrong!'
33| self.msg.setText( 'Password is %s' % result ) .
```

If the frame size isn't specified, and a call to the frame.pack() method is used to determine the initial size of the application, space won't be allocated for the message field. This choice would require the ActionListener method (i.e., the enter() method on lines 26-33) to adjust the size of the application frame so the message text is visible. It's much easier to specify the frame size when the frame is constructed.

The JPasswordField instance, as shown on lines 19-22, uses the first parameter to specify the number of characters that are allowed. Notice that you don't have to do anything special to specify an echo character. If you don't like the default echo character, you can use the setEchoChar() method, the echoChar attribute, or the keyword constructor argument to specify the character to be displayed when the user enters data.

Is There an Echo in Here? Using the Character-Obfuscation Property

Sometimes it might be convenient to allow the users to "turn off" the character-obfuscation property of a password field. If this is the case, you can change the echoChar property of the field to have a value of zero (i.e., \x00 or chr(0)). When this is the case, any text in the field will be displayed "in the clear." To enable obfuscation, you simply specify a non-zero echoChar value. The

PasswordDemo2.py script file uses a button to demonstrate one way that this might be performed.

The important part of this script is the showHide() method, which acts as the event handler for the application button. Initially, this button has a value of Show and can be used to display the user input in the clear. Listing 7-2 shows that when the button is activated, the method uses the current text associated with the button to determine how to proceed. When the button text is Show, the password field echoChar attribute is set to zero and the text of the button changes to Hide. When the button text is Hide, the password field echoChar attribute is set to its original value and the text of the button is reset to Show.

Listing 7-2. showHide() Method from PasswordDemo2.py

```
def showHide( self, event ) :
button = event.getSource()
if button.getText() == 'Show' :

self.pwd.setEchoChar( chr( 0 ) ) button.setText( 'Hide' )
else :
self.pwd.setEchoChar( self.echoChar ) button.setText( 'Show' )
```

Figure 7-2 shows some sample images from the PasswordDemo2.py script. The first image shows what happens when text is entered. The next image shows the result of using the Show button. Notice how the button value has been changed and the password input field is now “in the clear.”[3] The next image shows the result of using the Submit button to verify the user input, and the final image shows how the Hide button can be used to obfuscate the password text field again.

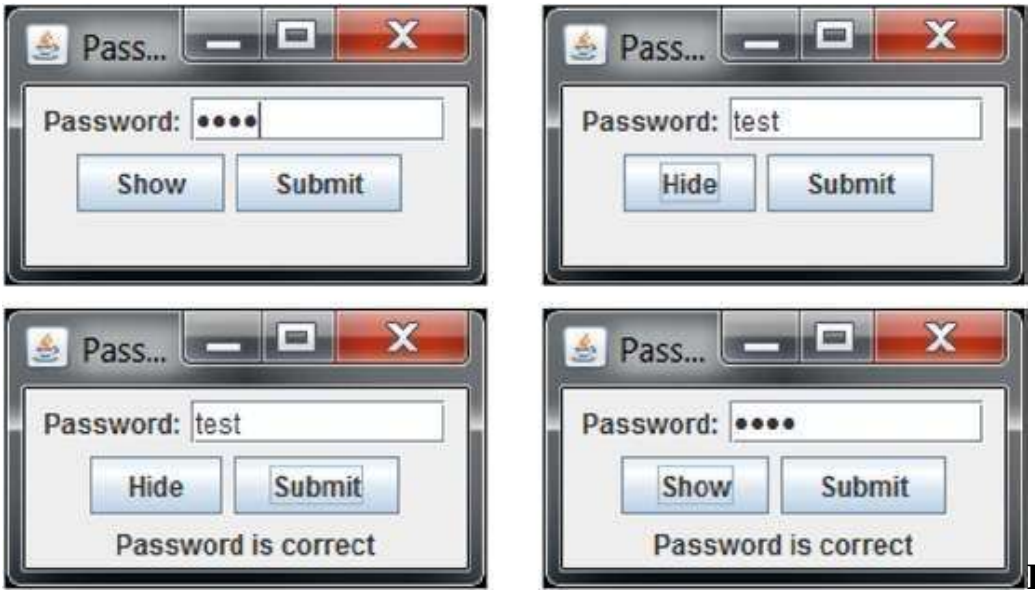


Figure 7-2. *PasswordDemo2.py sample output*

[3]It is interesting to note that even though the password text is visible, you still can't copy its value to the system's clipboard.

The getPassword() Method

Reading the Javadoc for the JPassword class should help you understand that even though two getText() methods exist for the underlying text field, they have been deprecated and shouldn't be used. Each getText() method description includes the following statement:

■ **Note** For security reasons, this method is deprecated. Use the getPassword method instead.

Some interesting things should be noted about the differences between the getText() methods and the getPassword() method. One of the most obvious differences is that the return type of the getText() methods is String, whereas the return type of the getPassword() method is a character array. The primary reason for this is security. Strings are immutable and will stick around in memory until a garbage collection cycle can free the storage. A character array, on the other hand, can be replaced or overwritten immediately.

The event.getActionCommand() Method

What's so important about the event handler that it warrants a separate section? JPasswordField has JTextField as its base class. Because of this, it inherits some

properties from that class that could be possible security concerns. The following statement comes from the JTextField Javadoc page:

JTextField will use the command string set with the setActionCommand method if it’s not null; otherwise, it will use the text of the field as a compatibility with java.awt.TextField.

What does it mean for your application? It should be a warning that if you aren’t careful, the actionCommand attribute of the JPasswordField instance may be a password string, which might be another possible security exposure. So it is a best practice to provide a non-null actionCommand value for your password field instance. Listing 7-3 shows how easily this can be done using the actionCommand keyword attribute on the constructor call. When this is done, using the Enter key is indistinguishable from using the Submit button as far as the event handler is concerned.

Listing 7-3. Defining a Non-Null actionCommand Attribute Value self.pwd = frame.add(

```
JPasswordField(
10,
actionCommand = 'Submit', actionPerformed = self.enter

)
)
```

The JPasswordField Event Handler

You finally get to the ActionListener event handler method that is used to process the user-supplied password. You can see an example of this routine in lines 26-33 of Listing 7-1. As mentioned earlier, the data type returned by calling the getPassword() method is a Java array.

The simplest way to check the user-supplied password against the required password value is to create a Java array of the appropriate type and value and compare it with the value returned by the getPassword() method. To do so, you can use the jarray module. If you are unfamiliar with the array method in the jarray module,[4] it exports two functions for the creation of Java arrays for Jython scripts.[5] The exported functions are explained in Table 7-1.

Table 7-1. *The jarray Module Functions* **Function Signature**
array(sequence, typeCode) zeros(length, typeCode)

Description
Returns a Java array containing the values initialized using the specified sequence. Returns a Java array containing zero (or null) values of the specified length.

The possible values for the typeCode argument of these functions are listed in Table 7-2.

Table 7-2. *typeCode Values Used by the jarray Module* typeCode **Character Value Data Type of the Returned Array** 'b' byte
'c' char
'd' double
'f' float
'h' short
'i' int
'l' long
'z' Boolean

Using this information allows you to better understand the expression on line 29 in Listing 7-1. Listing 7-4 demonstrates this even more clearly. From this image, you can see just how easy it is to create a Java array using a string value to initialize the array.

Listing 7-4. Creating a Java array

```
wsadmin>import jarray
wsadmin>
wsadmin>jarray.array( 'test', 'c' ) array(['t', 'e', 's', 't'], char) wsadmin>
```

Using functions from the jarray module allows you to simplify the testing of the value returned by the JPasswordField getPassword() method call. So, you need to be able to compare the value entered by the users against the appropriate password value. Unfortunately, you also need to be able to view the user password in this simple application. Please don’t use this kind of technique—i.e., hard-coded passwords—in your applications.

[4]Seehttp://www.jython.org/javadoc/org/python/modules/jarray.html. [5]Table 7-2

only lists the function signatures that have a typeCode argument for simplicity's sake. Converting jarray Values to Strings

What if you need a Jython string in your script? Can't you use the toString() method to convert the array value to a string? Unfortunately, the data types returned by the jarray array() function and the JPasswordField getPassword() method are, in fact, of type org.python.core.PyArray. Listing 7-5 demonstrates this.

Listing 7-5. Java array type

```
wsadmin>import jarray
wsadmin>from javax.swing import JPasswordField wsadmin>
wsadmin>result = jarray.array( 'test', 'c' ) wsadmin>type( result )
<jclass org.python.core.PyArray at 1926329041> wsadmin>
wsadmin>pwd = JPasswordField( 'test' )
wsadmin>type( pwd.getPassword() )
<jclass org.python.core.PyArray at 1926329041> wsadmin>
```

So how do you convert this Java array of characters into a string that can be passed to one of the wsadmin scripting object methods? Listing 7-6 shows a couple of ways to do this. The first builds a simple array of the individual characters, and then uses the string join() method to return a string formed by concatenating the characters of the array with an empty string between each. The second shows how simple this operation can be when list comprehension is used instead.

Listing 7-6. Converting a Java array of characters to a string

```
wsadmin>result = []
wsadmin>for ch in pwd.getPassword() :
wsadmin> result.append( ch )
wsadmin>
wsadmin>str( result )
"['t', 'e', 's', 't']"
wsadmin>''.join( result )
'test'
wsadmin>
wsadmin>
```

```
wsadmin>result = []
wsadmin>for ch in pwd.getPassword() :
wsadmin> result.append( ch )
wsadmin>
wsadmin>''.join( result )
'test'
wsadmin>
wsadmin>result = ''.join( [ ch for ch in pwd.getPassword() ] ) wsadmin>result
'test'
wsadmin>
```

Why do you need to worry about this? All you have to do is not set the ActionCommand string, and you will be able to retrieve the password as a string when the users press Enter, right? Not really. What happens when your application has multiple input fields, such as a user ID as well as a password? What about when you want to have two password fields for verification purposes? What happens when you also need or want to have a button? Then your script will have to use the JPasswordField getPassword() method to retrieve the Java array of characters. Then, before you can pass it to a WebSphere scripting object, you need some way to easily convert it from a Java array of characters into a Jython string. All this really means is that you need to know how to perform this type of conversion.

Choosing from a List

One nice technique for allowing users to provide input in your application is to provide a list of values and allow the users to select one. The Swing component that provides this kind of choice is the JComboBox. If you are interested in what that looks like, take a look at the sample output shown in Figure 7-3.

Figure 7-3. *ComboBoxDemo sample output*
There are some things that you should know about a JComboBox instance:

• The JComboBox class doesn’t include an actionPerformed attribute that can be used as keyword argument in the constructor call. So you must either use the addActionListener() method call or the actionListener keyword argument to identify the ActionListener instance to be used.

• One of the side effects of this decision is that the application class has to be a descendent of the ActionListener class, and it must have an event handler method named actionPerformed(). An example of this can be seen on lines 9, 21, and 24 in Listing 7-7.

• It is important to realize that only one ComboBox item can be selected at a time. Multiple items cannot be selected using this component.

• The ActionListener event handler code can use the event.getSource() method to obtain a reference to the JComboBox instance. This technique allows you to access the ComboBox object easily, an example of which is shown on lines 25 and 26 in Listing 7-7. An alternative is to have the application keep an object instance reference to the JComboBox object.

Listing 7-7. ComboBox Class

```
9|class ComboBoxDemo( java.lang.Runnable, ActionListener ) : 10| def run( self
) :
11| frame = JFrame(
12| 'ComboBoxDemo',
13| size = ( 200, 100 ),
14| layout = FlowLayout(),
15| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 16| )
17| frame.add( JLabel( 'Pick one:' ) )
18| choices = 'The,quick,brown,fox,jumped'.split( ',' ) 19| choices.extend(
'over,the,lazy,spam'.split( ',' ) ) 20| ComboBox = frame.add( JComboBox(
choices ) ) 21| ComboBox.addActionListener( self )
22| self.msg = frame.add( JLabel() )
23| frame.setVisible( 1 )
24| def actionPerformed( self, event ) :
25| ComboBox = event.getSource()
```

26| msg = 'Selection: ' + ComboBox.getSelectedItem() 27| self.msg.setText(msg)

Editing a ComboBox

One of the questions that can come up when you are looking at the JComboBox class is whether it can be edited. The simple answer is yes. There is an editable attribute that you can set to allow users to enter a value that isn't on the list. Figure 7-4 shows some sample output of the EditableComboBox.py script.

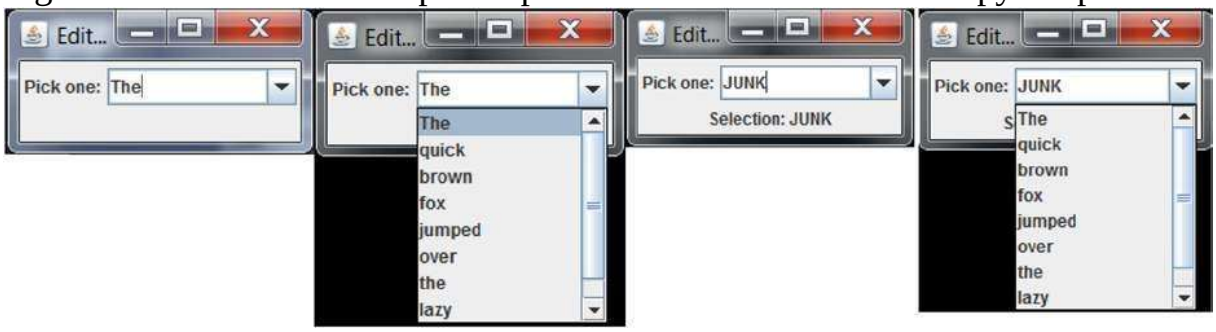


Figure 7-4. *EditableComboBox sample output*

What do you need to do to allow users to enter off-list values? Interestingly enough, you only have to set the editable attribute on the JComboBox instance. Listing 7-8 shows just how simple it is to do so.

Listing 7-8. Making a ComboBox Editable

```
ComboBox = frame.add(
JComboBox(
self.choices,
editable = 1

)
)
```

There is one important thing to note, however. Even though the event handler method can obtain a user value that isn't in the original list, the list of items doesn't change. So, if the users want to use the same value again, they will have to re-enter it. This should make you wonder if there is a way to add and remove items from the list.

Using the DynamicComboBox

There is also a DynamicComboBox, but it requires a bit more effort and code to use it well. First, take a quick look at some sample output from this sample

application, and then you can take a look at and learn about the instructions required to generate the desired results.

Figure 7-5 shows how the list of items in the ComboBox can be removed and added to completely replace the list. What does it take to completely replace the items in the list? The next three listings show methods from the DynamicComboBox.py script.

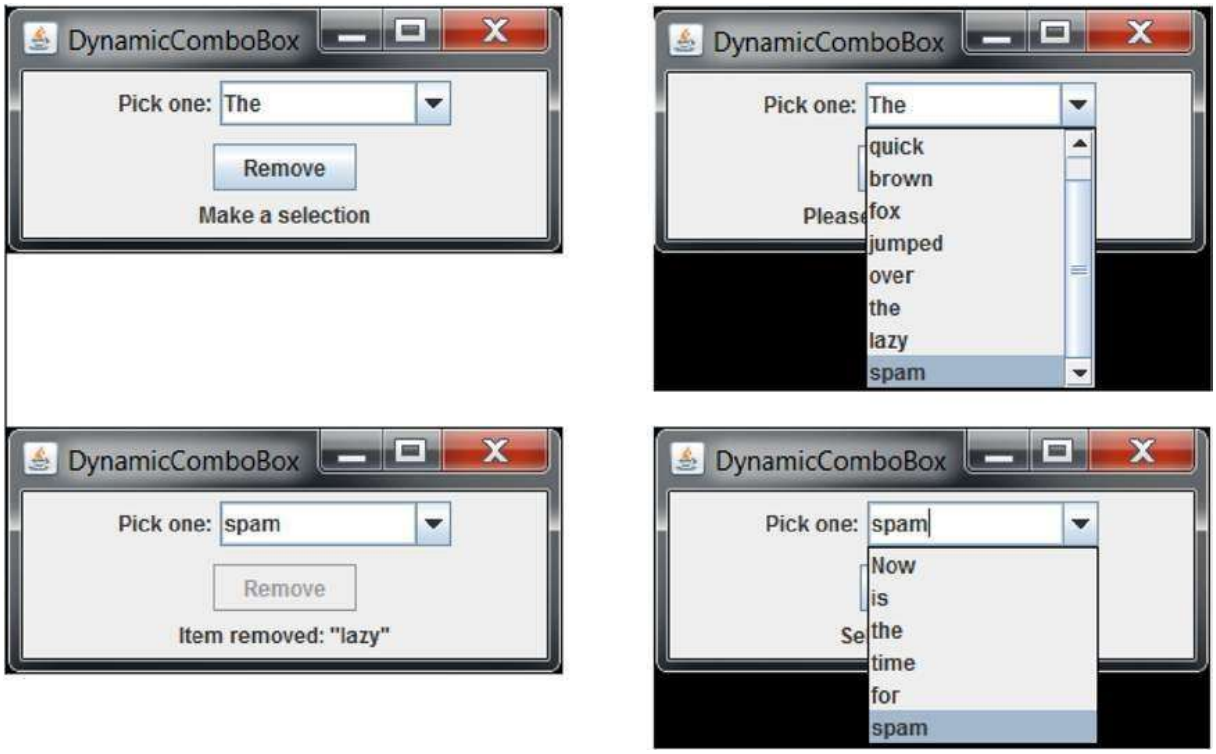


Figure 7-5. *DynamicComboBox sample output*

Listing 7-9 shows the run() method, which creates the various Swing components, places them on the application, and assigns the appropriate ActionListener event handler for the JComboBox and the new Remove button. This is where you'll see the initial list of nine ComboBox items (i.e., lines 22 and 23).

Listing 7-9. DynamicComboBox run() Method

```
12|class DynamicComboBox( java.lang.Runnable, ActionListener ) :
13| def run( self ) :
14| self.frame = frame = JFrame(
15| 'DynamicComboBox',
```

```
16| size = ( 310, 137 ),
17| layout = BorderLayout(),
18| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
19| )
20| panel = JPanel()
21| panel.add( JLabel( 'Pick one:' ) )
22| self.choices = 'The,quick,brown,fox,jumped'.split( ',' )
23| self.choices.extend( 'over,the,lazy,spam'.split( ',' ) )
24| self.ComboBox = ComboBox = JComboBox(
25| self.choices,
26| editable = 1
27| )
28| ComboBox.addActionListener( self )
29| panel.add( ComboBox )
30| frame.add( panel, BorderLayout.NORTH )
31| panel = JPanel()
32| self.RemoveButton = JButton(
33| 'Remove',
34| actionPerformed = self.remove
35| )
36| panel.add( self.RemoveButton )
37| frame.add( panel, BorderLayout.CENTER )
38| panel = JPanel( alignmentX = Component.CENTER_ALIGNMENT )
39| self.msg = panel.add( JLabel( 'Make a selection' ) )
40| frame.add( panel, BorderLayout.SOUTH )
41| frame.setVisible( 1 ) .
```

One thing that might surprise you is the use of JPanel instances. The first is created on line 20. Then, you add a label (line 21) and a ComboBox (line 29) to it. Finally, this panel is added to the frame on line 30. This allows both the label and the ComboBox to be kept together as a group of components, and then this panel can be positioned on the top of the frame using the BorderLayout.NORTH constant, as shown on line 30.

The creation of a second panel, on line 31, may surprise you. If you didn’t do this and simply added the button to the frame using the BorderLayout.CENTER constant, the button would increase to fill the available space, thus making it too large. By placing the button in a panel, as on line 36, the panel can be sized to fill the available space, while leaving the button at its preferred size in the

middle of the panel.

Then you create yet another panel, on line 38. This time you use the alignmentX keyword assignment to configure the horizontal alignment accordingly. If you chose to simply add the message label (e.g., self.msg on line 39) to the BorderLayout.SOUTH position instead, the message label would be aligned to left and would look out of place.

The actionPerformed() method, shown in Listing 7-10, is the ActionListener event handler that is associated with the ComboBox by the statement on line 28 in Listing 7-9. This method is invoked when someone uses a mouse button or a keyboard event to make a selection on the ComboBox. It is especially interesting to note that the item returned by the call to the getSelectedItem() method, as shown on line 44, might in fact return a value that doesn't currently exist in the list of items currently associated with the ComboBox. This is only possible when the ComboBox editable attribute is true, as you can see on line 26 in Listing 7-9.

Listing 7-10. DynamicComboBox actionPerformed() Method

```
42| def actionPerformed( self, event ) :
43| cb = self.ComboBox
44| item = cb.getSelectedItem().strip()
45| items = [
46| cb.getItemAt( i )
47| for i in range( cb.getItemCount() )
48| ]
49| if item :
50| if item not in items :
51| cb.addItem( item )
52| self.RemoveButton.setEnabled( 1 )
53| msg = 'Selection: "%s"' % item
54| self.msg.setText( msg )
55| else :
56| cb.setSelectedIndex( 0 )
```

This event handler is responsible for determining if the specified item is valid, and whether or not it currently exists on the list. If it doesn't exist, a call is made to the addItem() method (line 51) to add it to the list of items currently associated with the ComboBox. Why do you call the setEnabled() method when

an item is added? Because it is possible for the remove method, shown in Listing 7-11, to reduce the number of items on the ComboBox to one. At this point, the Remove button should be disabled.

Listing 7-11. DynamicComboBox remove() Method

```
57| def remove( self, event ) :
58| cb = self.ComboBox
59| index = cb.getSelectedIndex()
60| item = cb.getSelectedItem()
61| try :
62| cb.removeItem( item )
63| self.msg.setText( 'Item removed: "%s"' % item )
64| except :
65| self.msg.setText( 'Remove request failed' )
66| self.RemoveButton.setEnabled( cb.getItemCount() > 1 )
```

The remove() method, shown in Listing 7-11, is associated with the button, as shown on line 34 in Listing 7-9. This method is invoked only when the user selects the Remove button, and the event handler routine is responsible for removing the current ComboBox item. It is also responsible for disabling the button when the number of items on the ComboBox list is reduced to one.[6]

[6]The expression (i.e., cb.getItemCount() > 1) will be true (i.e., 1) when more than one item is present on the list; this keeps the button enabled. When the list only has one item, the expression is false (i.e., 0) and the button is disabled.

Formatted Text Fields

There are times when it is important to control the way information is displayed by a text field. For situations such as this, Swing provides the JFormattedTextField class[7] and the Format class hierarchy.[8]

A formatted text field allows you to specify the kinds of values that are appropriate for a specific field and determine how these values are displayed. What kind of formatting can you apply? Well, you can format a text field as a number (e.g., as an integer, a floating point, or as currency). You can also specify a pattern to identify how a value should appear (e.g., a date, Social Security Number, or even a telephone number).

Take a quick look at a sample application that uses the various instances of formatted numbers.[9] Figure 7-6 shows the output of this simple application.

Figure 7-6. *FormattedTextFieldDemo sample output*

What does it take to generate the output? You might be surprised to see just how easy it is. Listing 7-12 contains the code from the FormattedTextFieldDemo.py script used to produce the output shown in Figure 7-6.[10]

Listing 7-12. FormattedTextFieldDemo Class

```
class FormattedTextFieldDemo( java.lang.Runnable ) :
def addFTF( self, name ) :
pane = self.frame.getContentPane()
pane.add( JLabel( name ) )
pane.add(

JFormattedTextField(
eval( 'NumberFormat.' + name ),
value = 12345.67890,
columns = 10

)

)
def run( self ) :
self.frame = frame = JFrame(
'FormattedTextFieldDemo',
layout = GridLayout( 0, 2 ),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
self.addFTF( 'getInstance()' ) self.addFTF( 'getCurrencyInstance()' )
self.addFTF( 'getIntegerInstance()' ) self.addFTF( 'getNumberInstance()' )
self.addFTF( 'getPercentInstance()' ) frame.pack()
frame.setVisible( 1 )
```

[7]
Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JFormattedTextField.htr
[8]Seehttp://docs.oracle.com/javase/8/docs/api/java/text/Format.html.
[9]Note: You'll revisit the JFormattedTextField class in Chapter 14.
[10]Note: The eval() function, as shown in line 15, is normally discouraged. However, using it in this example allowed the addFTF() method call to be greatly simplified and much shorter.

Looking closely at this script, you can see that the format of the value to be displayed in each field is defined by the NumberFormat instance that is used to instantiate each formatted text field.
The generated output may make you wonder about the kind of control that is provided by the NumberFormat class. Looking at the NumberFormat Javadoc,[11] you see that you can modify the attributes in Table 7-3. By the way, this table shows the initial/default settings for each of these attributes for the specified NumberFormat instance type.

Table 7-3. *NumberFormat Instance Type Attributes*
NumberFormat Instance Type Integer Part Fraction Part Minimum Digits

getInstance() 1
getCurrencyInstance() 1
getIntegerInstance() 1
getNumberInstance() 1
getPercentInstance() 1

Maximum Digits Minimum Digits Maximum Digits MAXINT 0 3
MAXINT 2 2
MAXINT 0 0
MAXINT 0 3
MAXINT 0 0

How are you supposed to understand this information? Well, take a look at a row in the table, e.g., the one for the currency instances. It differs from the others in that these kinds of values have a minimum and maximum of two fractional digits. That's why the value in the currency row of Figure 7-6 appears as $12,345.68. If you execute that script and enter a value of $0 in the input field, it will be reformatted as $0.00, which makes sense when you look at the row in Table 7-3 specifying currency values. There will be a minimum of one digit

before and exactly two digits after the decimal point.

One important point to note about these types of constructors for the NumberFormat class is that the instance attributes are specific to the locale of the operating system on which the wsadmin script is being executed. If your application requires a specific locale formatting, it might be specified as a parameter on the instance constructor:

format = NumberFormat.getCurrencyInstance(Locale.FRENCH) [11]See http://docs.oracle.com/javase/8/docs/api/java/text/NumberFormat.html.

Using a JSpinner Text Field

The last of the text input fields to be discussed here makes use of the javax.swing.JSpinner class. What does the JSpinner object look like? Each JSpinner field has a text input field and two small buttons. Figure 7-7 shows a trivial JSpinner field displaying the days of the week.

It should be obvious that the text field in this particular example isn't wide enough to display all of the characters in the longest weekdays. First, take a look at the class used to display this sample output. Listing 7-13 shows the Spinner1 class from this script file.[12]

Figure 7-7. *Sample JSpinner text field*

Listing 7-13. Spinner1 Class

```
from java.text import DateFormatSymbols as DFS
...
class Spinner1( java.lang.Runnable ) :

def run( self ) :
```

```
frame = JFrame(
'Spinner1',
layout = FlowLayout(),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
daysOfWeek = [ dow for dow in DFS().getWeekdays() if dow ]
frame.add( JSpinner( SpinnerListModel( daysOfWeek ) ) )
frame.pack()
frame.setVisible( 1 )
```

Some DateFormatSymbols Methods

To understand this example, you need to understand a few concepts. Let's begin with the getWeekdays() method of the DateFormatSymbols module.[13] Listing 7-14 shows an interactive wsadmin session showing what is returned by the DateFormatSymbols' getWeekdays() and getMonths() methods.[14]

Listing 7-14. DateFormatSymbols methods

[12] Line 5 is included to show how an alias for the DateFormatSymbols was defined. This allowed line 17 to be short enough to be fit easily in the available space.

[13]Seehttp://docs.oracle.com/javase/8/docs/api/java/text/DateFormatSymbols.html

[14]There are many more methods provided by the DateFormatSymbols class that aren't covered in this book. Feel free to experiment with and use these other methods.

```
wsadmin>from java.text import DateFormatSymbols as DFS
wsadmin>
wsadmin>DFS().getWeekdays()
array(['', 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday'], java.lang. String)
wsadmin>
wsadmin>len( DFS().getWeekdays() )
8
wsadmin>DFS().getMonths()
wsadmin>array(['January', 'February', 'March', 'April', 'May', 'June', 'July',
'August', 'September', 'October', 'November', 'December', ''], java.lang.String)
```

It should be no surprise to you that the getWeekdays() and getMonths() methods return Java arrays. What might be a surprise to you—it certainly was to me—was the fact that the getWeekdays() method returns eight values and the getMonths() method returns 13 values.[15] That's why the Spinner1 class, as shown in Listing 7-13, uses list comprehension to process the results of calling the getWeekdays() method and returns the non-empty values that exist.

The JSpinner Class

Unlike the JComboBox, JSpinner instances do not display any kind of drop-down list of values. Only the current value is visible. The buttons on the field can be used to display the next or previous values. Why would you use a spinner instead of a ComboBox? Spinners are normally used when the number of valid items is too large to display. Does the use of the JSpinner class force the users to use the buttons? No, it doesn't.

Figure 7-8 shows the same Spinner1 application after various actions. You can see that the text portion of the spinner can be selected and user input (i.e., from the keyboard) can be used to filter or select the value to be displayed.

Figure 7-8. *Spinner value selections*

[15] This is *not* a bug, as is explained in the bug report found here: http://bugs.sun.com/bugdatabase/view_bug.do?bug_ id=4146173. It's a feature. Personally, I kind of like the explanation that exists in various places on the web that these eight-day weeks and 13-month years were created for managers and product planners to explain and justify the development schedules. But that's just a guess.

The various actions shown in Figure 7-9 are explained in Table 7-4.

Figure 7-9. *Default (numeric) spinner examples*

Table 7-4. *Spinner Value Selection, Explained*

Image	Description
1	Shows the application output after the window is slightly widened.
2	Shows how the text in the input field can be selected.
3	After typing W (and clicking on the window edge to cause the text input field to be widened), you can see that the first value matching the specified text has been selected. It is interesting to note how the W has been deselected.
4	After selecting the entire text in the field again.
5	After typing T, notice that Tuesday has been selected, even though it precedes Wednesday in the list of values.
6	After typing h, notice how Thursday has been selected. This tells you that the value that you typed is used to match the selected portion of the text field.

This raises a number of questions in my mind. What kinds of values can be displayed in a spinner field? Do they have to be alphabetic strings? Let's take a look at the JSpinner Javadoc[16] to find out.

According to that page, there are only two JSpinner constructors—the default (i.e., empty) and one that uses something called a SpinnerModel argument. The default spinner constructor shows that it can be used to display numeric (i.e., integer). The images in Figure 7-9 show how numeric spinner values can be selected.

From left, the first image shows the initial, or default, value of 0. The second shows how a million is displayed. The third shows the maximum integer value (i.e., java.lang.Integer.MAX_VALUE of 2,147,483,647). The final image shows the value shown when the button is used to display the next value. Hopefully, you aren't too surprised by the fact that a wraparound occurs, and the value that is displayed is the smallest integer value (i.e., java.lang.Integer.MIN_VALUE of -2,147,483,648).

One of the interesting things to note about this output is how the numeric values are formatted. From this output, you should quickly come to the realization that the text field portion of the spinner instance is a formatted text field.

The SpinnerModel Class

The SpinnerModel Javadoc[17] explains that this class is used for potentially unbounded values, which makes sense, especially looking back at the default spinner that you saw in Figure 7-9.

[16]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JSpinner.html.

[17]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/SpinnerModel.html.

One of the important points to note about the SpinnerModel is that it is an interface. Your applications can use one of your own classes based upon this interface class or one of the implementations provided by the Swing API, specifically SpinnerDateModel, SpinnerListModel, or SpinnerNumberModel. Looking back at Listing 7-13, you can see that the Spinner1 class uses the SpinnerListModel class. The constructor for this class was passed an array of strings. This particular example contains a short list of values, so a ComboBox may be a better choice in this type of situation. But it's your decision.

The SpinnerNumberModel class constructors have either zero or four parameters. The zero parameters constructor was used by the default JSpinner() constructor shown on line 15 of Listing 7-15.

Listing 7-15. Spinner2 Class

```
8|class Spinner2( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(
11| 'Spinner2',

12| layout = FlowLayout(),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
14| )
15| frame.add( JSpinner() )
16| frame.pack()
17| frame.setVisible( 1 )
```

This is the one that was used to display a selection of four billion values, as shown in Figure 7-9. The four-parameter variant of the SpinnerNumberModel class constructor identifies:

- The initial value to be displayed
- The minimum valid value

• The maximum valid value
• The stepSize, or increment value, to be used

The Spinner3 class that shows how this might be used is demonstrated in Listing 7-16.

Listing 7-16. Spinner3 Class

```
class Spinner3( java.lang.Runnable ) :
def run( self ) :
frame = JFrame(

'Spinner3',
layout = FlowLayout(),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
frame.add(
JSpinner(
SpinnerNumberModel(

0, # Initial value
-3141.59, # Minimum value
+3141.59, # Maximum value

3.14159 # stepSize
)

)
)
frame.pack()
frame.setVisible( 1 )
```

One of the interesting points to note about this application is that no value exists above the maximum or below the minimum specified values. This differs from the Spinner2 class, which uses a wraparound effect.

Figure 7-10 shows what the default SpinnerDateModel looks like, at least for my locale. The multiple images show that you can select different portions of the date in the text field and then use either the spinner buttons or the up and down keys on your keyboard to change the date.

Figure 7-10. *Default SpinnerDateModel examples*

For example, I set the current date to 3/1/00 and selected the day of the month part (i.e., the 1). Pressing the down key changes the date to 2/29/00 (i.e., Leap Day). How do you specify March 1, 2000 as the starting date for the spinner? Listing 7-17 shows one way to do this and is part of the Spinner5.py script.

Listing 7-17. Sample Use of SpinnerDateModel

```
frame.add(
JSpinner(

SpinnerDateModel(
Date( 2000, 2, 1 ), # zero origin month
None, # minimum
None, # maximum
Calendar.DAY_OF_MONTH # Ignored by GUI

)
)
)
```

One of the things to remember about this example is that even though a calendarField argument is specified, this does not automatically cause the specified field to be selected on the application window. You might be surprised to see the cursor located at the beginning or end of your input field, depending on your locale.

The JSpinner Editor

What if you don’t like the way that the value (e.g., Date) is shown in the spinner text field? To change it, you need to change the default spinner editor, which is based on the kind of SpinnerModel being used. Four spinner editors are provided by Swing:

• JSpinner.DateEditor—Used for SpinnerDateModel instances

• JSpinner.ListEditor—Used for SpinnerListModel instances
• JSpinner.NumberEditor—Used for SpinnerNumberModel instances
• JSpinner.DefaultEditor—A simple base class for more specialized editors

Figure 7-11 shows the sample output that is generated using a different date pattern used by the Spinner6 class shown in Listing 7-18.

Figure 7-11. *The Spinner6 output using a different date pattern*

Lines 27-30 show how you can specify a different date display pattern[18] for the formatted text field used by the spinner instance.

Listing 7-18. Spinner6 Class

```
10|from javax.swing import SpinnerDateModel
11|class Spinner6( java.lang.Runnable ) :
12| def run( self ) :
13| frame = JFrame(
14| 'Spinner6',
15| layout = FlowLayout(),
16| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
17| )
18| spinner = JSpinner(
19| SpinnerDateModel(
20| Date( 2000, 2, 1 ), # zero origin month
21| None, # minimum
22| None, # maximum
23| Calendar.DAY_OF_MONTH # Ignored by GUI
24| )
25| )
26| spinner.setEditor(
27| JSpinner.DateEditor(
28| spinner,
29| 'dd MMM yy'
30| )
31| )
32| frame.add( spinner )
33| frame.pack()
```

```
34| frame.setVisible( 1 )
```

This is just a simple example that should provide you with enough of a start for your applications.

Summary

This chapter explained many new input components that you can now utilize. By now, you should be getting a better feel for how Swing components can be used to provide user-friendly input choices for the users of your applications. In the next chapter, you will turn your attention to selectable input components.

[18] See http://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html.

Chapter 8

Selectable Input Components

So far, you have learned how to use simple buttons and a variety of text input fields. In this chapter, you'll take a look at using some other input components, ones that can be selected. All of the components in this chapter have two possible states, they are either selected or not. I'll start by describing toggle buttons, proceed to check boxes, and then to radio buttons. I will also discuss how to group these fields in order to make interesting user presentations.

Toggle Buttons

Simple push buttons are used to initiate an event of some kind. Occasionally, it can be useful for a button to have an associated state to convey additional information to the users. Toggle buttons perform this role. Thinking of this obvious difference makes me think of when I was little. The radio in my parents' car had buttons that you would press to select a station. It was quite obvious which of the buttons had been selected. You will learn how to group toggle buttons together so that only one button can be selected at a time.

Before you do that, though, you need to better understand toggle buttons and their states. The toggle button state can indicate whether the button has been selected/clicked or not. What does a toggle button look like? Figure 8-1 shows a simple application window containing a single toggle button, before and after it

is pressed. In order to accentuate the button's state, this application also changes the button's text based on its selected attribute.

Figure 8-1. *Sample toggle button application output*

What does it take to create a JToggleButton?[1] Listing 8-1 shows the ToggleButton class that instantiates a toggle button using the specified text and identifies the event handler to be invoked when the button state changes. The most significant difference between simple buttons and toggle buttons is the presence of the selected state. A push button, on the other hand, initiates an action when the button is clicked. Because of this, the event handler of the toggle button invokes the itemStateChanged() method of the ItemListener interface.[2] By comparison, the event handler for a JButton will invoke the actionPerformed() method of the ActionListener interface.[3] You can see this difference in the itemStateChanged keyword argument of the JToggleButton constructor in Listing 8-1.

[1] See http://docs.oracle.com/javase/8/docs/api/javax/swing/JToggleButton.html.
[2]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/event/ItemListener.html.
[3]Seehttp://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html.

Listing 8-1. ToggleButton Class

```
8|class ToggleButton( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(

11| 'Toggle Button',
12| layout = FlowLayout(),
13| size = ( 275, 85 ),
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| button = JToggleButton( # Make a toggle button
17| 'Off' , # Initial button text
```

```
18| itemStateChanged = self.toggle # Event handler
19| )
20| frame.add( button )
21| frame.setVisible( 1 )
22| def toggle( self, event ) :
23| button = event.getItem()
24| button.setText( [ 'Off', 'On' ][ button.isSelected() ] )
```

The toggle method, shown in lines 22-24 of Listing 8-1, uses the getItem() method of the event that caused the method to be called in order to determine which component had a state change. It wasn't necessary to do this in this simple application, because there is only one component that has this property. However, this example is useful when you have multiple components that share an event handler. In fact, you'll see this scenario in the next example.

Check Boxes

While reviewing the Javadoc for the JCheckBox class,[4] I found it kind of interesting to see that it was, in fact, based on the JToggleButton class that you just read about. This makes sense because toggle buttons and check boxes both have a simple on/off state and react when clicked. Using this knowledge, you can easily create an application based on the ToggleButton code that has some check boxes. Figure 8-2 shows the some sample output of this application.

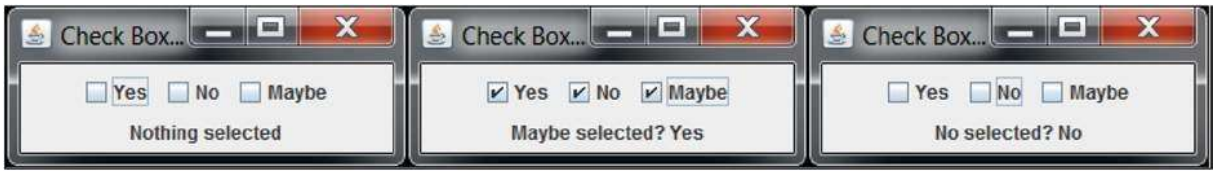


Figure 8-2. *Sample output for check boxes*

Listing 8-2 shows one way that this can be done.[5] A point of interest is how the event handler uses the getText() method (line 31) to retrieve the text of the check box that caused the state change method to be called. As mentioned earlier, this allows the check boxes to share the same event handler.

[4] Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/JCheckBox.html.

[5]Initially, I had the code to create and add each JCheckBox to the frame on a single line. Width limitations for these listings, however, meant that it would be best, at least for this example, to create and use the addCB() method in lines 9-

15 instead.

Listing 8-2. CheckBoxes Class

```
8|class CheckBoxes( java.lang.Runnable ) :
9| def addCB( self, pane, text ) :
10| pane.add(
11| JCheckBox(
12| text,
13| itemStateChanged = self.toggle
14| )
15| )
16| def run( self ) :
17| frame = JFrame(
18| 'Check Boxes',
19| layout = FlowLayout(),
20| size = ( 250, 100 ),
21| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 22| )
23| cp = frame.getContentPane()
24| self.addCB( cp, 'Yes' )
25| self.addCB( cp, 'No' )
26| self.addCB( cp, 'Maybe' )
27| self.label = frame.add( JLabel( 'Nothing selected' ) ) 28| frame.setVisible( 1 )
29| def toggle( self, event ) :
30| cb = event.getItem()
31| text = cb.getText()
32| state = [ 'No', 'Yes' ][ cb.isSelected() ]
33| self.label.setText( '%s selected? %s' % ( text, state ) )
```

■**Note** this example doesn’t use the selected parameter of the JCheckBox constructor, so each of the check boxes starts with the state as deselected.

Radio Buttons

One of the differences between check boxes and radio buttons is that radio buttons are most useful when they are grouped. This allows one, and only one, radio button in the group to be selected at a time. Take a look at the previous example again; this time, however, it uses radio buttons instead of check boxes.

Figure 8-3 shows sample output of this simple application, which is now using radio buttons instead of check boxes.

Figure 8-3. *Sample output for radio buttons*

Listing 8-3 shows the RadioButtons class that generates the output shown in Figure 8-3. What, if anything, do you notice about this class? Take a few moments to compare it to the CheckBoxes class in Listing 8-2. You should find very few differences between these two classes.

Listing 8-3. RadioButtons Class

```
9|class RadioButtons( java.lang.Runnable ) :
10| def addRB( self, pane, bg, text ) :
11| bg.add(
12| pane.add(
13| JRadioButton(
14| text,
15| itemStateChanged = self.toggle 16| )
17| )
18| )
19| def run( self ) :
20| frame = JFrame(
21| 'Radio Buttons',
22| layout = FlowLayout(),
23| size = ( 250, 100 ),
24| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 25| )
26| cp = frame.getContentPane()
27| bg = ButtonGroup()
28| self.addRB( cp, bg, 'Yes' )
29| self.addRB( cp, bg, 'No' )
30| self.addRB( cp, bg, 'Maybe' )
31| self.label = frame.add( JLabel( 'Nothing selected' ) ) 32| frame.setVisible( 1 )
33| def toggle( self, event ) :
34| text = event.getItem().getText()
35| self.label.setText( 'Selection: ' + text )
```

As mentioned earlier, you also need a ButtonGroup[6] to indicate the collection to which the new JRadioButton[7] object should be added. This ButtonGroup is created on line 27 and passed to the addRB() method on lines 28-30. So that shouldn't be too much of a surprise. Were you surprised that with one statement —in lines 11-18—you can:

• Create a radio button (line 13)
• Specify the associated text (line 14)
• Specify the ActionListener (i.e., itemStateChanged) event handler (line 15)
• Add the ActionListener to the specified pane (line 12)
• Add the ActionListener to the specified button group (line 11)

I found this quite nice, and I appreciate how easy it is to understand the code required to do this. Granted, the example isn't perfect, but it is very easy to replace an application component with a closely related one.

[6]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/ButtonGroup.html.

[7]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JRadioButton.html.

Toggle Buttons in a Button Group

In the first part of this chapter I mentioned how the radio buttons of my parents' car were similar to toggle buttons. Thinking of this made me wonder how difficult it would be to simulate this feature in a script. All you need to do is place a bunch of toggle buttons in a button group, right? Well, that's pretty much the case. There are a few extra things that you need to do (initialize one of the buttons as selected and display the correct message), but that's pretty much it. Figure 8-4 shows the sample output for this application.

Figure 8-4. *A group of toggle buttons*

What does it take to do this? Not much. In fact I think that you'll agree that Listing 8-4 looks very similar to Listing 8-3, where you used a button group to identify a group of radio buttons.

Listing 8-4. ButtonGroupDemo

```
class ButtonGroupDemo( java.lang.Runnable ) :
def addRB( self, pane, bg, text ) :
bg.add(

pane.add(
JToggleButton(
text,

selected = ( text == '1' ),
itemStateChanged = self.toggle
)
)

)
def run( self ) :
frame = JFrame(
'ToggleButton Group',
layout = FlowLayout(),
size = ( 265, 100 ),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE
)
cp = frame.getContentPane()
bg = ButtonGroup()
for i in range( 1, 6 ) :
self.addRB( cp, bg, `i` )
self.label = frame.add( JLabel( 'Selection: 1' ) )
frame.setVisible( 1 )
def toggle( self, event ) :
text = event.getItem().getText()
self.label.setText( 'Selection: ' + text )
```

Here’s a question for you. Do toggle buttons in a button group act any differently than those that aren’t in a button group? The answer is yes they do. Once a toggle button within a button group is selected, it can’t be deselected. In fact, this is the same behavior shown when radio buttons are used. Initially, you can have all of the radio buttons (or toggle buttons) in a button group deselected, but once one has been selected, one will always be selected.[8]

Summary

This chapter, even though it is fairly short, discusses and describes selectable input components. One of the interesting things that I found while investigating these components was the fact that all the selectable classes are based on the JToggleButton class. This is what led me to investigate using multiple toggle buttons within a button group. I hope that you find these an interesting addition to your collection of user interface components.

[8]Unless, of course, you have something like an event handler deselect everything in the group. **Chapter 9**

Providing Choices, Making Lists

You are likely to encounter situations where it would be nice to provide your users with a list of choices. For example, you've probably selected the name of the city where you live from a list. Maybe you want to build an application to keep track of the books or movies that you own. Fortunately, Swing provides the JList[1] component, which allows programmers to build and display lists of this sort. In this chapter, you learn how to build and display a list of items. You will also learn how to manipulate a list in the event handler method associated with another component, such as a button.

Making a List and Checking It Twice

How hard is it to make a list? Not very. Listing 9-1 shows how little code is required to create a list using a group of words from a string. Another version of this script, called List1a.py, is provided that shows you how to build a JList using a java.util.Vector.

Listing 9-1. The List1 Class class List1(java.lang.Runnable) :
def run(self) :
frame = JFrame(

'List1',
size = (250, 200),
defaultCloseOperation = JFrame.EXIT_ON_CLOSE

)
data = 'Now is the time for all good spam'.split(' ') frame.add(JList(data))
frame.setVisible(1)

What can you do with a default JList? Figure 9-1 shows that you can select one or more items on this list with no additional code; you simply need to press the Ctrl key while additional items are selected.

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JList.html.

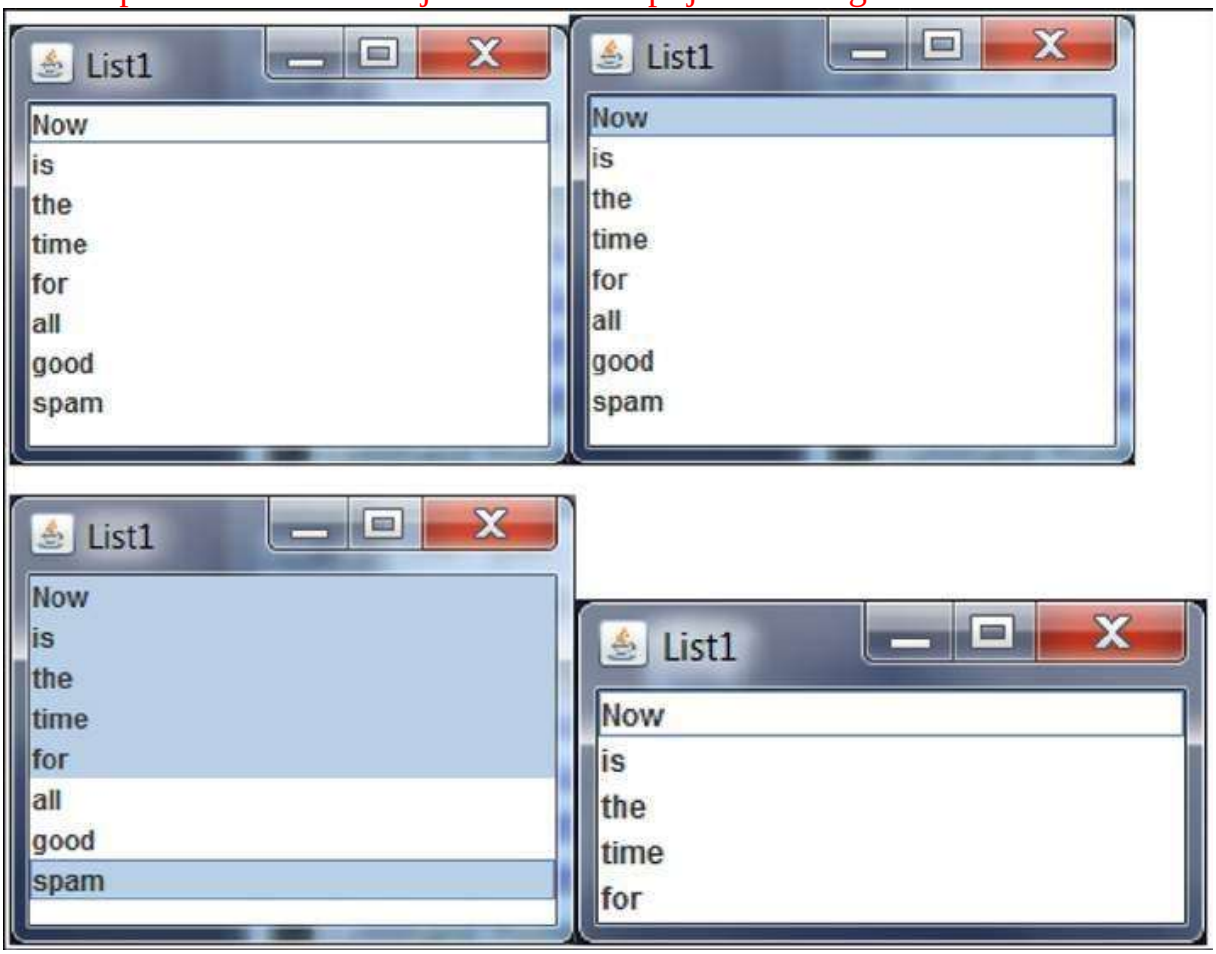


Figure 9-1. *List1's sample output*

Wow, that was easy, right? You're all done and can move on to the next topic, right? Not quite. There are lots of things that you need to consider when your applications deal with lists. For example, the last image shows one problem that occurs when the size of the frame isn't large enough to show all of the entries in the list. There is no indication that additional items exist. Even worse, if you use the cursor down arrow to move the selection down the list, you can select items that are not visible to the user. What can you do about that?

Optional Scroll Bars

If your list has a limited (small) number of items (such as the days of the week),

you might want to use something like a JComboBox[2] instead of a JList. Frequently, however, the number of items on the list won't easily fit in the application window. When this happens, you only need to wrap the JList instance within a JScrollPane.[3] This is so easy to do that it is hard to imagine why you wouldn't want to always put your JList instance in a scroll pane. Listing 9-2 shows just how easy this can be. Compare line 15 to line 14 in Listing 9-1.

Listing 9-2. Wrapping a JList in a Scroll Pane

```
7|class List2( java.lang.Runnable ) :
8| def run( self ) :
9| frame = JFrame(
10| 'List2',
```

[2]As discussed in section 7.2.
[3]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JScrollPane.html.

```
11| size = ( 250, 100 ),
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
13| )
14| data = 'Now is the time for all good spam'.split( ' ' )
15| frame.add( JScrollPane( JList( data ) ) )
16| frame.setVisible( 1 )
```

What does this do to the application? Well, if the list contains more items than can be displayed in the available area, vertical and/or horizontal scroll bars will be added, as needed, to allow the users to determine which parts of the available information they want to view. Figure 9-2 shows the output of the List2 application. It has space for fewer lines and therefore requires a vertical scrollbar.[4] Once the size of the frame is large enough to display the complete list, the vertical scrollbar automatically disappears.

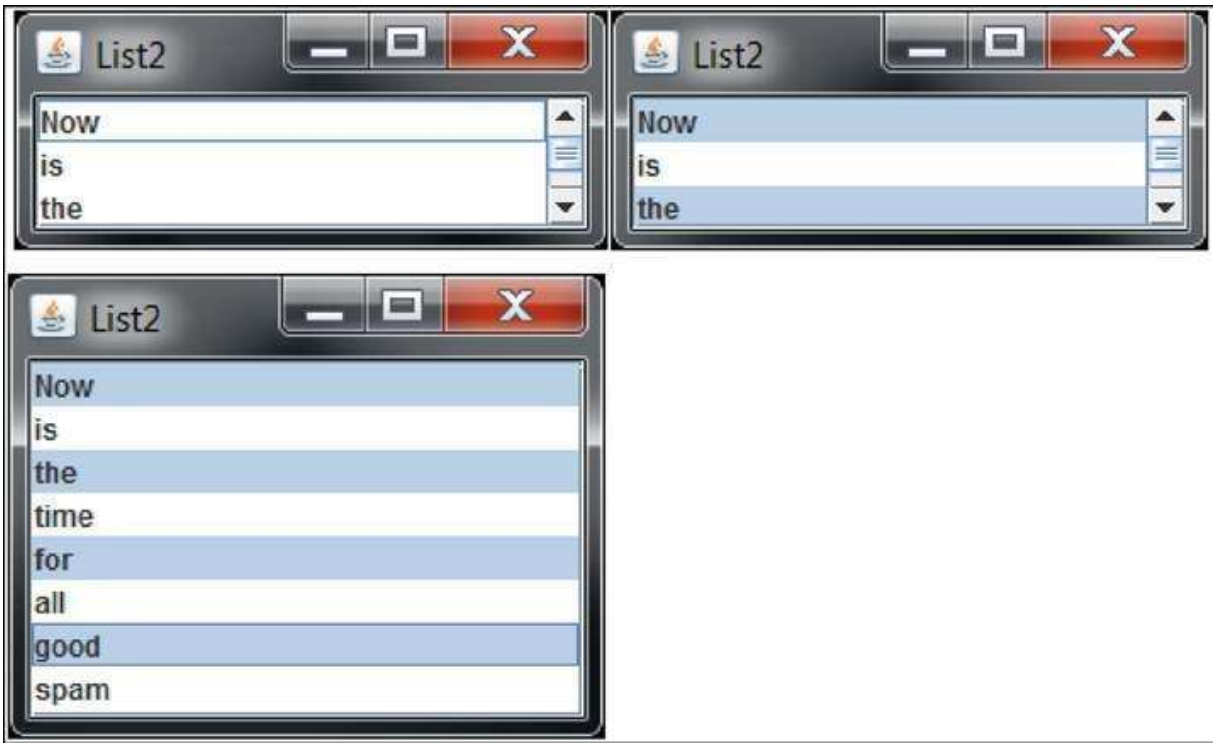


Figure 9-2. *List2's sample output*

The ScrollPane Viewport

The JScrollPane Javadoc includes a diagram that shows the relationship between the ScrollPane instance and the child component contained in it. One of the important concepts to note is that a JViewport[5] is created to determine the portion of the child component to be displayed.

■ **Tip** if you are interested in learning more about scroll panes, viewports, and scroll bars, i encourage you to take a look at the "how to Use scroll panes"[6] portion of the Java swing tutorials. this book doesn't delve into more detail about viewports, but they do show up in Chapter 12, where tables are discussed in detail.

[4] In case you are wondering, I simply held the Ctrl key and selected the odd list entries to produce these images.

[5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JViewport.html.

[6]See http://docs.oracle.com/javase/tutorial/uiswing/components/scrollpane.html.

Manipulating the List

You will frequently need your applications to manipulate a list in some way. For example, you may want to add, remove, or replace items on a list. To do this, you need to use the interface provided by the list model. What's a list model? It is the component that actually contains the list contents and provides methods that allow your applications to manipulate the list. If one is not passed to the JList constructor, a DefaultListModel[7] will be created to for you.[8] You previously saw this separation of the component and its data in Chapter 7, which discussed the SpinnerModel class.

Counting List Items

Let's take a quick look at how a list model can be used to count the number of times a specific value occurs on the list. Figure 9-3 shows some sample output for the List3 application. The first image (from the left) shows the initial look of the application. The second image shows what the application looks like after you enter some text into the input field (i.e., "the") and click the Count button. Note how the input field is cleared and the message portion of the application is updated to reflect the number of occurrences of the specified word.

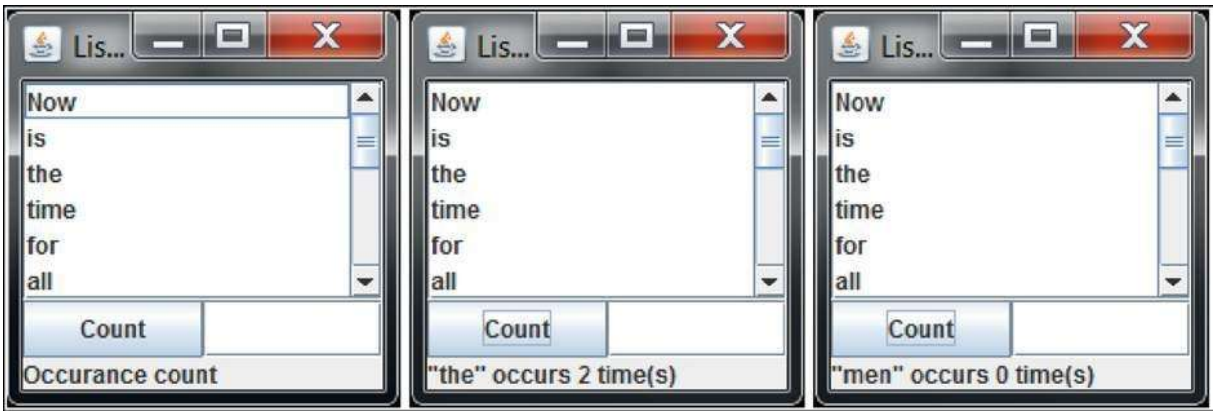


Figure 9-3. *List3's sample output*

Listings 9-3 and 9-4 show the List3 class produces the output shown in Figure 9-3. Lines 21-25 show how the list of words is created, again using a simple array of strings. Lines 26-32 show how the list is wrapped in a scroll pane and added to the top of the application window using the BorderLayout.NORTH position constant.

Listing 9-3. List3's run() Method

```
13|class List3( java.lang.Runnable ) :
```

```
14| def run( self ) :
15| frame = JFrame(
16| 'List3',
17| size = ( 200, 200 ),
18| layout = BorderLayout(),
19| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
20| )
```

[7] See http://docs.oracle.com/javase/8/docs/api/javax/swing/DefaultListModel.html.
[8]This book discusses the use of the DefaultListModel, not the AbstractListModel, the ListModel, or the more interesting SortedListModel class discussed at http://java.sun.com/developer/technicalArticles/J2SE/Desktop/sorted_jlist/

```
21| data = (
22| 'Now is the time for all good spam ' +
23| 'to come to the aid of their eggs'
24| ).split( ' ' )
25| self.info = JList( data )
26| frame.add(
27| JScrollPane(
28| self.info,
29| preferredSize = ( 200, 110 )
30| ),
31| BorderLayout.NORTH
32| )
33| panel = JPanel( layout = GridLayout( 0, 2 ) )
34| panel.add(
35| JButton(
36| 'Count',
37| actionPerformed = self.count
38| )
39| )
40| self.text = panel.add( JTextField( 10 ) )
41| frame.add( panel, BorderLayout.CENTER )
42| self.msg = JLabel( 'Occurance count' )
43| frame.add( self.msg, BorderLayout.SOUTH )
44| frame.setVisible( 1 )
```

The middle portion of the application, containing the Count button and text input fields, is created and positioned on the application window in lines 33-41. Finally, the status message (a label field) is created and placed on the bottom of the application window in lines 42 and 43.

Listing 9-4 shows the count() method, which is identified as the button's actionListener event handler, on line 37 in Listing 9-3, when the button is created. It is here that you need to use the list model to count the number of times a specified value occurs in the associated list.[9]

Listing 9-4. List3's count() Method

```
45| def count( self, event ) :
46| word = self.text.getText()
47| model = self.info.getModel()
48| occurs = 0
49| for index in range( model.getSize() ) :
50| if model.getElementAt( index ) == word :
51| occurs += 1
52| self.msg.setText(
53| '"%s" occurs %d time(s)' %
54| ( word, occurs )
55| )
56| self.text.setText( " )
```

[9]You wouldn't want to use this technique on an unbounded list. And, if your list was large, you would want the list model processing to be performed on a separate thread (i.e., by a SwingWorker class instance).

Limiting the Selectable Items

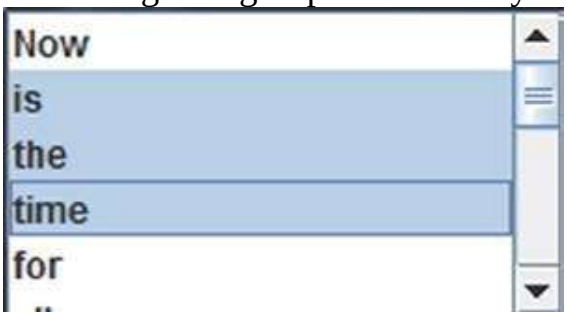
Other list-manipulation actions include adding and removing items to and from the list. Before you take a look at what these operations require, think about it for a moment. When you want to add an item to the list, where do you want to add it? Should it be added to the beginning of the list, at the end, or somewhere in the middle? You also might need to consider if multiple occurrences of a value should be allowed on the list.

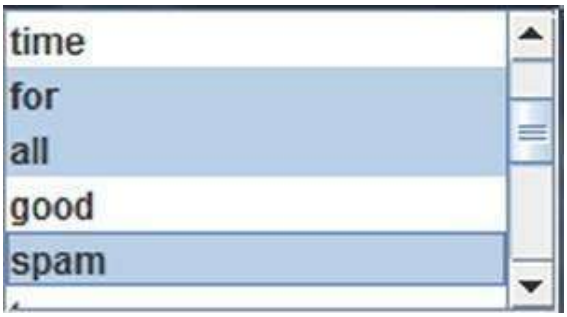
Do you want to allow multiple items to be selected? This might make sense if you want to allow the users to remove multiple items with a single button click,

but this might not always make sense. For example, if you want to allow the users to select an item on the list, and then allow a new item to be added, either before or after the selected item, then it doesn’t make sense to allow multiple items to be selected at one time.

Earlier, you saw that the default list properties allow you to select multiple list items. How do you disable this? The JList class includes a selection mode attribute that can define the number of list items that can be selected. Table 9-1 describes the list selection mode values.

Table 9-1. *List Selection Mode Examples*

Selection Mode	Description
SINGLE_SELECTION	Either zero or one item may be selected.
 SINGLE_INTERVAL_SELECTION	Only one contiguous group of items may be selected.
	
MULTIPLE_INTERVAL_SELECTION	The default mode allows multiple groups of items to be selected.



In Java applications, you would normally use the JList setSelectionMode() method to identify the mode to be used. In Jython, you can use this method call or you can use the selectionMode keyword argument when you create the JList object. Listing 9-5 shows an example of how easily this can be done using the constructor keyword argument

Listing 9-5. Using the List Selection Mode Keyword Argument

```
|26|        self.info = JList(                                          |
|27|            data,                                                  |
|28|            selectionMode = ListSelectionModel.SINGLE_SELECTION    |
|29|        )                                                           |
```

Reacting to List-Selection Events

Can you think of a reason that your application needs to know when an item on a list has been selected? What kinds of things might you want to do with the selected item? You might want to remove it, modify it, or insert a new item before or after the selected item. You might also want to initiate some more complex operation based on the user selection. It's all up to you.

If no item is selected, what kinds of things might you want to be able to do? What about adding items to the list? Where might you be able to add an item if you have to reference it (a selected item)? What about adding a new item first or last on the list? Those options make sense, don't they?

The reason I'm asking these questions is to help you think about things that you might want to consider as you are creating your own applications.
Do list selection events exist to be monitored? Sure. The challenge is trying to figure out what you want to do when a list selection event occurs. What do you want to happen when the users select a list item? It depends on what your application looks like and what it does.
What if you partition the application output to have the scrollable list and a text input field on top and some buttons below? Figure 9-4 shows one way to do this.

Figure 9-4. *List4's sample output*

Listing 9-6 shows how easily this can be done using Jython. To partition the application window, the frame uses a BorderLayout, with the scrollable pane containing the list instance positioned using the BorderLayout.NORTH constant (line 35), and another pane that uses a GridLayout to position the text field and buttons within it (lines 37-43). Then, this pane is positioned on the application frame using the BorderLayout.SOUTH constant (line 44).

Listing 9-6. List4's Class

```
14|class List4( java.lang.Runnable ) :
15| def run( self ) :
16| frame = JFrame(
17| 'List4',
18| size = ( 200, 222 ),
19| layout = BorderLayout(),
20| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
21| )
22| data = (
23| 'Now is the time for all good spam ' +
24| 'to come to the aid of their eggs'
25| ).split( ' ' )
```

```
26| self.info = JList(
27| data,
28| selectionMode = ListSelectionModel.SINGLE_SELECTION
29| )
30| frame.add(
31| JScrollPane(
32| self.info,
33| preferredSize = ( 200, 100 )
34| ),
35| BorderLayout.NORTH
36| )
37| panel = JPanel( layout = GridLayout( 0, 2 ) )
38| self.text = panel.add( JTextField( 10 ) )
39| panel.add( self.button( 'First' ) )
40| panel.add( self.button( 'Last' ) )
41| panel.add( self.button( 'Before' ) )
42| panel.add( self.button( 'After' ) )
43| panel.add( self.button( 'Remove' ) )
44| frame.add( panel, BorderLayout.SOUTH )
45| frame.setVisible( 1 )
46| def button( self, text ) :
47| return JButton( text, actionPerformed = self.insert )
48| def insert( self, event ) :
49| todo = event.getActionCommand()
50| word = self.text.getText()
51| print '%s: "%s"' % ( todo, word )
```

Now that you have an idea how the application will look, you might be able to figure out what you want to happen when a list item is selected. You can also think about changing the appearance of the application if you don't like the way it looks. What if you had the buttons and text field on the left and the list of items on the right? What would that look like? It takes very little time and effort to figure this out. Figure 9-5 shows the result of making these changes.

Figure 9-5. *List5's sample output*

Looking at Listing 9-7, you can see how few changes were needed to produce this output. Additionally, you now have an event handler that is called when a list selection operation has occurred (lines 35 and 51-54).
This method currently only produces output on the console, but it provides you with information about when a selection event occurs, as well as informs you that you can use the available information to determine whether a list item has been selected. You can do this by looking at the Javadoc for the ListSelectionListener class[10] and verifying that you only need to identify the valueChanged() method (see line 35).

Listing 9-7. List5's Class Reacting to List Selection Events

```
13|class List5( java.lang.Runnable ) :
14| def run( self ) :
15| frame = JFrame(
16| 'List5',
17| size = ( 200, 220 ),
18| layout = GridLayout( 1, 2 ),
19| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
20| )
21| panel = JPanel( layout = GridLayout( 0, 1 ) )
22| panel.add( self.button( 'Remove' ) )
23| panel.add( self.button( 'First' ) )
```

```
24| panel.add( self.button( 'Last' ) )
25| panel.add( self.button( 'Before' ) )
26| panel.add( self.button( 'After' ) )
27| self.text = panel.add( JTextField( 10 ) )
28| frame.add( panel )
29| data = (
30| 'Now is the time for all good spam ' +
31| 'to come to the aid of their eggs'
32| ).split( ' ' )
33| self.info = JList(
34| data,
35| valueChanged = self.selection,
36| selectionMode = ListSelectionModel.SINGLE_SELECTION 37| )
38| frame.add(
39| JScrollPane(
40| self.info,
41| preferredSize = ( 200, 100 )
42| )
43| )
44| frame.setVisible( 1 )
45| def button( self, text ) :
46| return JButton( text, actionPerformed = self.insert ) 47| def insert( self, event
) :
48| todo = event.getActionCommand()
49| word = self.text.getText()
50| print '%s: "%s"' % ( todo, word )
51| def selection( self, e ) :
52| index = e.getSource().getSelectedIndex()
53| if not e.getValueIsAdjusting() :
54| print 'selected %d' % index
```

[10]Seehttp://docs.oracle.com/javase/8/docs/api/javax/swing/event/ListSelectionLis

Reacting to User (Text) Input

All right, you've given some thought as to how you want the application to look and decided that you want to do something when a list item is selected. What do you want to happen next? Wouldn't it be neat to have the application's buttons be enabled or disabled based on the user input? For example, it makes sense to enable the Remove button when a list item is selected. What about the other

buttons? When do you want them to be enabled? What if you only enable the buttons related to list insertion when the user specifies some text in the text input field?

The InputMethodListener class[11] looks easy to use, right? But will it solve the problem at hand? Unfortunately, it is not going to work in this scenario. The List6.py script contains an attempt to add an InputMethodListener event handler.[12] Unfortunately, when you try to execute this script you get an AttributeError exception. Why? Because the addInputMethodListener() method is provided by the javax.swing.text.JTextComponent class, not the Runnable or InputMethodListener classes.

Listing 9-8. List6.py Script Interesting Lines

```
15|from java.awt.event import InputMethodListener
16|class List6( java.lang.Runnable, InputMethodListener ) : | ...
29| self.addInputMethodListener( self )
| ...
69| def caretPositionChanged( self, e ) :
70| print 'caretPositionChanged() :', e
71| def inputMethodTextChanged( self, e ) :
72| print 'inputMethodTextChanged() :', e
```

Listing 9-8 shows how easy it is to make these kinds of changes. Why do I encourage you to look at failures? Well, there are a few very important reasons to do so:

[11] See http://docs.oracle.com/javase/8/docs/api/java/awt/event/InputMethodListener.htm

[12]Please note that the line in List6.py that corresponds to line 29 in Listing 9-8 is commented out. To see the exception you need to remove the “#” in column one of that statement before executing the script.

- To try new things
- To learn how to fail gracefully
- To learn from your mistakes

It only takes a few moments to search for AWT and Swing methods related to events that might be called when the users enter text. It only required you to insert or modify seven lines of code. Was this a huge investment of time and effort? No, and what you learned was priceless.

Is there another way to use an InputMethodListener? Certainly, you can create a class to implement the InputMethodListener interface and see if that works. The List6a.py script contains this attempt. Listing 9-9 shows the modified lines from List6.py to demonstrate this iteration.

Listing 9-9. List6a.py Script, Unique Lines Only

```
15|from java.awt.event import InputMethodListener
16|class IML( InputMethodListener ) :
17| def caretPositionChanged( self, e ) :
18| print 'caretPositionChanged() :', e
19| def inputMethodTextChanged( self, e ) :
20| print 'inputMethodTextChanged() :', e
21|class List6a( java.lang.Runnable ) :
| ...
33| self.text = panel.add( JTextField( 10 ) )
34| self.text.addInputMethodListener( IML() )
```

Unfortunately, if you test this script, you will see that modifying the input field does not generate any output. Additional research shows that another approach might be more viable. For this iteration, you see what you need to use a DocumentListener instead of an InputMethodListener. This example is found in List6b.py; the modified lines are shown in Listing 9-10.

Listing 9-10. List6b.py Script, Unique Lines Only

```
15|from javax.swing.event import DocumentListener
16|class DL( DocumentListener ) :
17| def changedUpdate( self, e ) :
18| print 'changedUpdate() :', e
19| def insertUpdate( self, e ) :
20| print 'insertUpdate() :', e
21| def removeUpdate( self, e ) :
22| print 'removeUpdate() :', e
23|class List6b( java.lang.Runnable ) :
| ...
35| self.text = panel.add( JTextField( 10 ) )
36| self.text.getDocument().addDocumentListener( DL() ) It is interesting to note that in order to have a DocumentListener for the input field, you can use the
```

getDocument() method of the JTextField class to identify the actual Document object that is associated with this input field. If you test this iteration, as shown in Figure 9-6, you can see that events are being generated and the corresponding DocumentListener methods are being called. The particular events were the result of typing a character and then using the Backspace key to delete it from the input field.

This approach might not be the best way to monitor changes to an input field, especially with something simple like the JTextField, which is used in these examples. Fortunately, there is another approach that you can investigate.

```
insertUpdate() : [javax.swing.text.GapContent$InsertUndo@...]
removeUpdate() : [javax.swing.text.GapContent$RemoveUndo@...]
```

Figure 9-6. *List6b's DocumentListener sample output*

For this example, you are going to look at what happens when you add a listener to the input field for KeyListener events.[13]

```
text: "b"
text: "ba"
text: "bac"
text: "baco"
text: "bacon"
```

Figure 9-7 shows that you can monitor the text input field using a KeyListener keyReleased() method call. Listings 9-11 and 9-12 show the List7 class used to produce this output.

```
text: "b"
text: "ba"
text: "bac"
text: "baco"
text: "bacon"
```

Figure 9-7. *List7's sample output*

Listing 9-11. List7's run() Method

```
16|class List7( java.lang.Runnable, KeyListener ) :
17| def run( self ) :
18| frame = JFrame(
19| 'List7',
20| size = ( 200, 220 ),
```

```
21| layout = GridLayout( 1, 2 ),
22| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
23| )
24| panel = JPanel( layout = GridLayout( 0, 1 ) ) 25| self.buttons = {}
26| for name in 'First,Last,Before,After,Remove'.split( ',' ) : 27| self.buttons[
name ] = panel.add( self.button( name ) ) 28| self.text = panel.add(
29| JTextField(
30| 10,
31| keyReleased = self.typed
32| )
33| )
34| frame.add( panel )
35| data = (
36| 'Now is the time for all good spam ' +
37| 'to come to the aid of their eggs'
38| ).split( ' ' )
39| model = DefaultListModel()
40| for word in data :
41| model.addElement( word )
42| self.info = JList(
43| model,
44| valueChanged = self.selection,
45| selectionMode = ListSelectionModel.SINGLE_SELECTION 46| )
47| frame.add(
48| JScrollPane(
49| self.info,
50| preferredSize = ( 200, 100 )
51| )
52| )
53| frame.setVisible( 1 )
```

[13]See http://docs.oracle.com/javase/8/docs/api/java/awt/event/KeyListener.html.
Listing 9-11 shows the run() method and Listing 9-12 shows the rest of the methods from this class.

Listing 9-12. The Remaining List7 Class Methods

```
54| def button( self, text ) :
55| return JButton(
56| text,
```

```
57| enabled = 0,
58| actionPerformed = self.doit
59| )
60| def doit( self, event ) :
61| todo = event.getActionCommand()
62| word = self.text.getText().strip()
63| List = self.info
64| pos = List.getSelectedIndex()
65| print '%s: "%s" pos: %d' % ( todo, word, pos )
66| if todo == 'Remove' :
67| List.getModel().remove( pos )
68| self.buttons[ todo ].setEnabled( 0 )
69| def selection( self, e ) :
70| if e.getValueIsAdjusting() :
71| si = e.getSource().getSelectedIndex() 72| self.buttons[ 'Remove' ].setEnabled(
si > -1 ) 73| def typed( self, e ) :
74| print 'text: "%s"' % self.text.getText().strip()
```

Figure 9-8 shows the progress in the application output using the next iteration of the script. This one is found in the List8.py script file. You can see which buttons are enabled before any text is entered, after some text is entered, after a single item is selected, and finally after a button is used to insert the new word on the list.

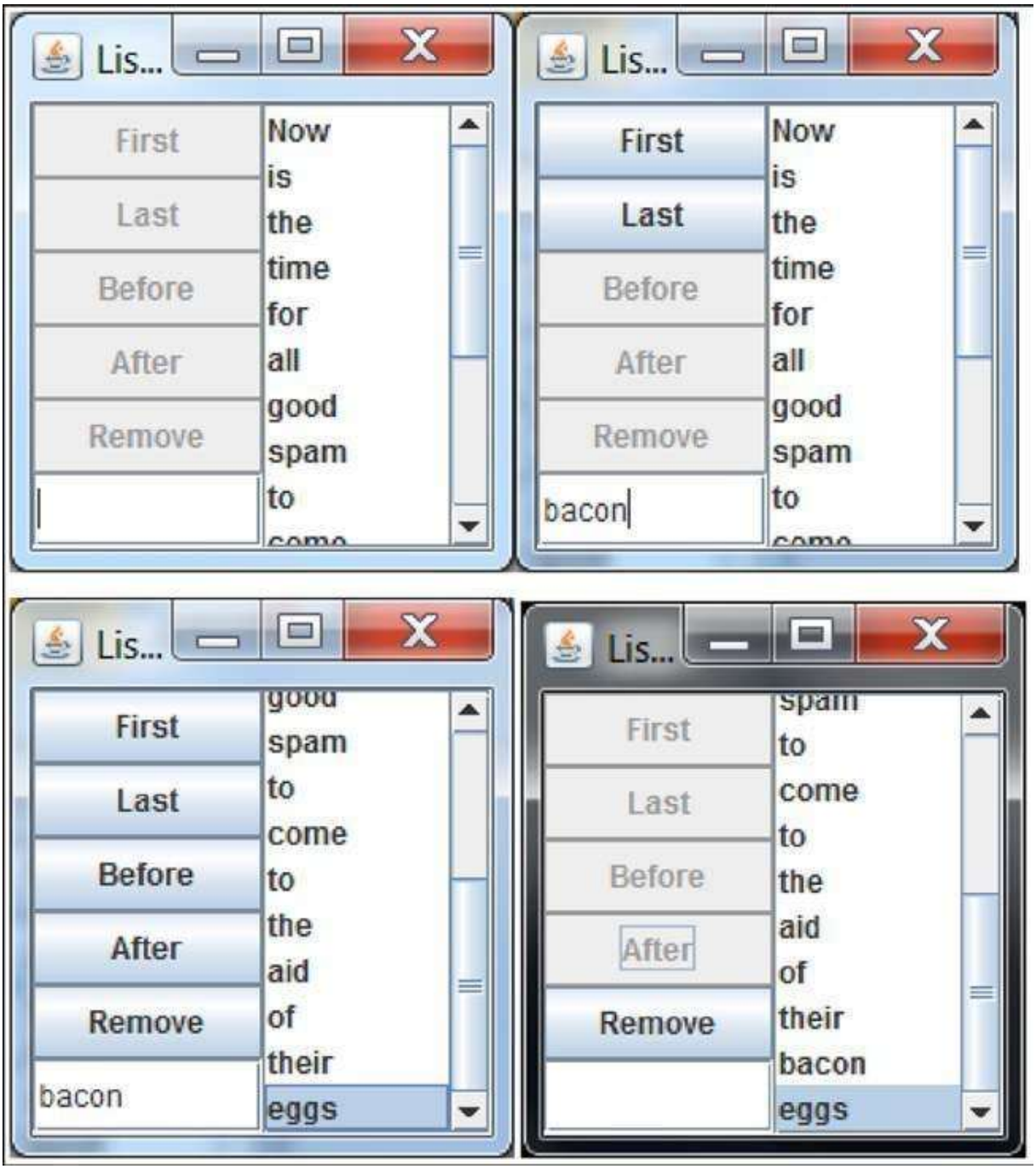


Figure 9-8. *List8's sample output*

You can look at the iterations of the application and see how small changes have been made with each. Finally, you get to the final iteration, which is found in the List10.py script file. This version includes the small improvement of putting the removed item text in the text input field. This allows you to do:

1. Select a the first item.
2. Click the Remove button (thus putting the item value into text field).
3. Click the Last button to move the first item to the end of the list.

Listing 9-13 contains the List10 class methods that allow your application to do these things. The run() method is not included because it's almost identical to the run() method found in Listing 9-11. The List10 class does, however, include some interesting properties. You might not agree that these are advantages. Specifically, List10 takes advantage of the named keyword arguments to use the textCheck() method as:

- A KeyListener keyReleased() method
- A ListSelectionListener valueChanged() method
- A simple textCheck() class

Java purists may take offense at this flexibility. However, I think that most Jython programmers can see the value provided by the use of keyword argument lists in this type of application.

Listing 9-13. Remaining List10 Methods

```
57| def doit( self, event ) :
58| todo = event.getActionCommand()
59| word = self.text.getText().strip()
60| List = self.List
61| pos = List.getSelectedIndex()
62| model = List.getModel()
63| if todo == 'Remove' :
64| self.text.setText( List.getSelectedValue() )
65| model.remove( pos )
66| else :
67| if todo == 'First' :
68| model.insertElementAt( word, 0 )
69| elif todo == 'Last' :
70| model.insertElementAt( word, model.getSize() )
71| elif todo == 'Before' :
72| model.insertElementAt( word, pos )
73| else :
74| model.insertElementAt( word, pos + 1 )
75| self.text.setText( '' )
```

```
76| self.textCheck()
77| def textCheck( self, e = None ) :
78| word = self.text.getText().strip()
79| index = self.List.getSelectedIndex()
80| for name in 'First,Last'.split( ',' ) :
81| self.buttons[ name ].setEnabled( len( word ) > 0 )
82| for name in 'Before,After'.split( ',' ) :
83| self.buttons[ name ].setEnabled(
84| len( word ) > 0 and index > -1
85| )
86| self.buttons[ 'Remove' ].setEnabled( index > -1 )
```

One of the most impressive things about this example is that it demonstrates how easy it is to quickly produce applications. Being able to produce a rapid prototype is enormously valuable to programmers because they can then easily manipulate and interact with the application and decide what they like and dislike about the way it looks, as well as the way it responds to user input.

Summary

This chapter covered how to display and manipulate lists of items with an emphasis on iterating the application to test various representations of components on the frame as well as using a variety of listeners to improve your understanding of how each might or might not enhance the usability of the application. You saw just how quickly and easily a script can be modified to test a new approach so that you can discard those that don’t fit your specific needs.

Note: One point that this chapter didn’t cover was the fact that once modifications have been made to the list of values, the application should retrieve the current list’s contents using the ListModel methods. **Chapter 10**

Menus and MenuItems

Menus are one of the most common input mechanisms that applications use. When was the last time that you worked with a graphical application that didn’t include some sort of menu structure and hierarchy?

The main reason for creating and using menus in your applications is that it conveys to the users some of the actions that they can perform with the

application. So this is all about setting expectations and communicating at least some of the things that the applications can do.

This chapter shows how easily you can add a menu to your applications and how to make your applications react when the user selects a menu item.

The JMenu Class Hierarchy

You may not realize it, but menus are a kind of button. This makes a lot of sense when you think about what happens when you click a button and when you click on a menu item. Each causes some kind of event that needs an event handler to perform the desired action. There are differences, though. If there weren't, you wouldn't need another class. Take a moment to think about it. How would you describe a menu? I think that you'll agree that they are almost always a collection of individual words that convey information to the users. Each word or menu entry can be selected and then displays a related sub-menu or performs some kind of response. A button, on the other hand, is almost always used to initiate some kind of action or elicit a specific response.

You can see how menus and buttons are similar by looking at the JMenu class[1] Javadoc or by using the classInfo routine first mentioned in Chapter 4. The classInfo hierarchy for the JMenu class is shown in Listing 10-1.

Listing 10-1. JMenu Class Hierarchy

```
wsadmin>classInfo( JMenu )
javax.swing.JMenu
| javax.swing.JMenuItem
| | javax.swing.AbstractButton
| | | javax.swing.JComponent
| | | | java.awt.Container
| | | | | java.awt.Component
| | | | | | java.lang.Object
| | | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer | | | | | java.io.Serializable | | | | java.io.Serializable
| | | java.awt.ItemSelectable
| | | javax.swing.SwingConstants | | javax.accessibility.Accessible | | javax.swing.MenuElement
| javax.accessibility.Accessible | javax.swing.MenuElement
wsadmin>
```

[1]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenu.html.

Unlike regular buttons, or even toggle buttons, the result of activating a menu is to cause the associated list of items to be displayed. So how do you add a JMenu to the application? Since you are using a JFrame for almost all of your applications, let's take another quick look at the JFrame hierarchy, with a focus on the methods that contain the text "menubar"

Listing 10-2 shows this hierarchy. One important point to note is that the javax.swing.JFrame class has JMenuBar getters and setters, and the java.awt.Frame class has MenuBar getters and setters.[2]

Listing 10-2. JFrame MenuBar Methods

```
wsadmin>classInfo( JFrame, meth = 'menubar' ) javax.swing.JFrame
getJMenuBar, setJMenuBar

| java.awt.Frame
> getMenuBar, setMenuBar
| | java.awt.Window
| | | java.awt.Container
| | | | java.awt.Component
| | | | | java.lang.Object
| | | | | java.awt.image.ImageObserver | | | | | java.awt.MenuContainer
| | | | | java.io.Serializable
| | | javax.accessibility.Accessible
| | java.awt.MenuContainer
| javax.swing.WindowConstants
| javax.accessibility.Accessible
| javax.swing.RootPaneContainer
wsadmin>
```

What does this mean? For one, it means you need to be careful. Listing 10-3 shows what I'm talking about. Since there are getters and setters for JMenuBars as well as MenuBars, you need to be certain that you remember to include the *J*, or you'll get an exception like the one shown in lines 12 and 13.

Listing 10-3. Adding a MenuBar to the JFrame

```
1|wsadmin>from javax.swing import JFrame, JMenuBar, JMenu
2|wsadmin>
```

```
3|wsadmin>f = JFrame( 'Frame Title' )
4|wsadmin>b = JMenuBar()
5|wsadmin>m = b.add( JMenu( 'Help' ) )
6|wsadmin>f.setMenuBar( m )
7|WASX7015E: Exception running command: "f.setMenuBar( m )";
8| exception information:
9| com.ibm.bsf.BSFException: exception from Jython: 10|Traceback (innermost
last):
11| File "<input>", line 1, in ?
12|TypeError: setMenuBar(): 1st arg can't be coerced to 13| java.awt.MenuBar
14|wsadmin>
```

[2]The Swing JMenuBar methods are used exclusively in this book.

Let's take a quick look at how you can create an empty, yet colorful, menu bar so that you can more easily identify where it's placed. Figure 10-1 shows where the blue JMenuBar is positioned on the JFrame content pane. By selecting a colorful background, you don't have to add any items to the menu to see where it.

Figure 10-1. *Menu1 sample output with a blue JMenuBar*

Listing 10-4 shows how easily you can do this.

■ **Note** You have to identify the preferred JMenuBar size (line 17), since it contains no JMenu entries. Otherwise no space would be allocated for the menu and it wouldn't be visible.[3]

Listing 10-4. The Menu1 Class Using Color to Show an Empty JMenuBar

```
7|class Menu1( java.lang.Runnable ) :
8| def run( self ) :
9| frame = JFrame(
10| 'Menu1',
11| size = ( 200, 125 ),
12| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 13| )
```

```
14| frame.setJMenuBar(
15| JMenuBar(
16| background = Color.blue,
17| preferredSize = ( 200, 25 )
18| )
19| )
20| frame.setVisible( 1 )
```

How will the application look with a couple of menu entries? Let's see.... Since the JMenuBar has a blue background color, let's make the menu entries use a color that contrasts well. Figure 10-2 shows a first attempt at doing this.

[3]I encourage you to comment out line 17 and test this statement yourself. Don't forget, however, to comment out the trailing comma on the previous line as well.

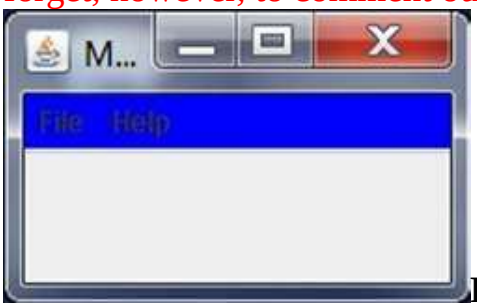


Figure 10-2. *Menu2 sample output with a JMenuBar and entries*

That's not quite what I expected. What happened to the white foreground color? Listing 10-5 shows the Menu2 class.

Listing 10-5. The Menu2 Class Specifying Foreground and Background Colors

```
9|class Menu2( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'Menu2',
13| size = ( 200, 125 ),
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
15| )
16| menuBar = JMenuBar(
17| background = Color.blue,
18| foreground = Color.white
19| )
20| fileMenu = JMenu( 'File' )
21| fileMenu.add( JMenuItem( 'Exit' ) )
```

```
22| menuBar.add( fileMenu )
23| helpMenu = JMenu( 'Help' )
24| helpMenu.add( JMenuItem( 'About' ) )
25| menuBar.add( helpMenu )
26| frame.setJMenuBar( menuBar )
27| frame.setVisible( 1 )
```

The reason the menu items don't show up well is that you specified the foreground color for the JMenuBar, *not* for the menu entries. If you specify the color of the menu items, you can see the expected contrast in colors. Figure 10-3 demonstrates this expected contrast.

Figure 10-3. *Menu3 sample output with colorful JMenuBar entries*

Listing 10-6 shows the Menu3 class being used to generate this output. It also shows you that building a hierarchy of menu entries can make your code a bit more challenging to read.

Listing 10-6. Menu3 Class with Contrasting Menu Entries

```
9|class Menu3( java.lang.Runnable ) :
10| def run( self ) :
11| frame = JFrame(
12| 'Menu3',
13| size = ( 200, 125 ),
14| defaultCloseOperation = JFrame.EXIT_ON_CLOSE 15| )
16| menuBar = JMenuBar( background = Color.blue ) 17| fileMenu = JMenu(
'File', foreground = Color.white ) 18| fileMenu.add( JMenuItem( 'Exit' ) )
19| menuBar.add( fileMenu )
20| helpMenu = JMenu( 'Help', foreground = Color.white ) 21| helpMenu.add(
JMenuItem( 'About' ) )
22| menuBar.add( helpMenu )
23| frame.setJMenuBar( menuBar )
24| frame.setVisible( 1 )
```

Looking closely at Listing 10-6, you can find three different kinds of menu-related objects: JMenuBar: Created on line 16[4]•
• JMenu: Created on lines 17 and 20
• JMenuItem: Created on lines 18 and 21[5]

Looking again at Figure 10-3, you can see most of these distinct entries and you can use this to better understand these different menu classes. You can also see how the color scheme of the selected entries differs from the ones specified. The good news is that you don't have to describe this new scheme because it's already provided by the Java Swing classes. This is another example of the Swing library making your life as a GUI developer that much easier.

Unfortunately, not everything is this easy. Here is where you might get into a little bit of potential confusion based on names and terminology. If you ask non-programmers what the row of words across the top of an application is called, they are likely to call it a "menu," which makes sense.

For Swing developers, however, the row of words corresponds with the Java, or Jython, JMenuBar class instance and the words themselves are instances of the JMenu objects. In order to minimize confusion, this book consistently uses the Java Swing terminology. I'll call the group of words or terms the JMenuBar, whereas the individual objects are called JMenu entries.

As you can see, creating the JMenuBar and the JMenu entries, as well as the corresponding JMenuItems, requires a fair amount code. It's common practice to have a method that is responsible for the creating menu bars and their entries and items. This method makes your application constructor (or run() method) simpler, and therefore easier to read, understand, and maintain.

[4]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenuBar.html. [5]See http://docs.oracle.com/javase/8/docs/api/javax/swing/JMenuItem.html.

Reacting to Menu-Related Events

One of the nice things about menu items being a kind of abstract button is that you already know how to tell the "button" what to do when it is pressed. You can use the verbose Java addActionListen() method call or the actionPerformed keyword argument to identify the method to be called.

Consider this: It’s common practice for Java Swing developers to have the application class descend from the ActionListener class. This is frequently done using the syntax shown in Listing 10-7.

Listing 10-7. Java Class Implementing ActionListener

```
public myClass implements ActionListener { ...
JMenuItem menuItem = new JMenuItem( "Spam" );
menuItem.addActionListener( this );
...
publick void actionPerformed( ActionEvent e ) {

...
}
...

}
```

Using this technique can make the actionPerformed() method more complex than it really needs to be. If every button and menu item uses the same actionPerformed() method, this event handler needs to figure out which kind of event was used to initiate the call to the actionPerformed() method. Listing 10-8, on the other hand, shows how easy it can be, using Jython, to have a simple routine for each menu item action listener event handler.

Not only does this approach simplify each event handler, it also allows each method to have a name that is more appropriate for the event being handled.

Listing 10-8. Using the actionPerformed Keyword Argument

```
8|class Menu4( java.lang.Runnable ) :
9| def run( self ) :
10| frame = JFrame(

11| 'Menu4',
12| size = ( 200, 125 ),
13| defaultCloseOperation = JFrame.EXIT_ON_CLOSE
14| )
15| menuBar = JMenuBar()
16| fileMenu = JMenu( 'File' )
17| exitItem = fileMenu.add(
18| JMenuItem(
```

```
19| 'Exit',
20| actionPerformed = self.exit
21| )
22| )
23| menuBar.add( fileMenu )
24| helpMenu = JMenu( 'Help' )
25| aboutItem = helpMenu.add(
26| JMenuItem(
27| 'About',
28| actionPerformed = self.about
29| )
30| )
31| menuBar.add( helpMenu ) 32| frame.setJMenuBar( menuBar ) 33|
frame.setVisible( 1 ) 34| def about( self, event ) : 35| print 'Menu4.about()' 36|
def exit( self, event ) : 37| print 'Menu4.exit()'
38| sys.exit()
```

Granted, in Java you don’t need the ActionListener actionPerformed() method to determine which event caused the routine to be called. You could make each JMenuItem use an anonymous ActionListener class implementation. Unfortunately, this approach is much more challenging to read, use, and maintain than the Jython technique used in the Menu4 class in Listing 10-8.

Using Radio Buttons on a Menu

Up to this point, the menu items have only included text. There are times, however, when it makes sense to use radio buttons on a menu to identify only selection. I didn’t have to look far to find an example of this. I often use the Notepad++ text editor, which has an Encoding menu item. The sub-menu on this entry includes five radio button menu items that identify the encoding. There is a special class for this in the Swing hierarchy, called JRadioButtonMenuItem.[6]

As you saw with radio buttons in Chapter 8, you need to be able to group all of the JRadioButtonMenuItem instances together so that the Swing framework can enforce one selected item limitation. Fortunately, you can use the same ButtonGroup class for the menu items that you used for radio buttons.