



More-Sophisticated Behavior

Using library classes to implement
some more advanced functionality



Main concepts to be covered

- Using library classes
- Reading documentation



The Java class library

- Thousands of classes
- Tens of thousands of methods
- Many useful classes that make life much easier
- Library classes are often inter-related
- Arranged into packages



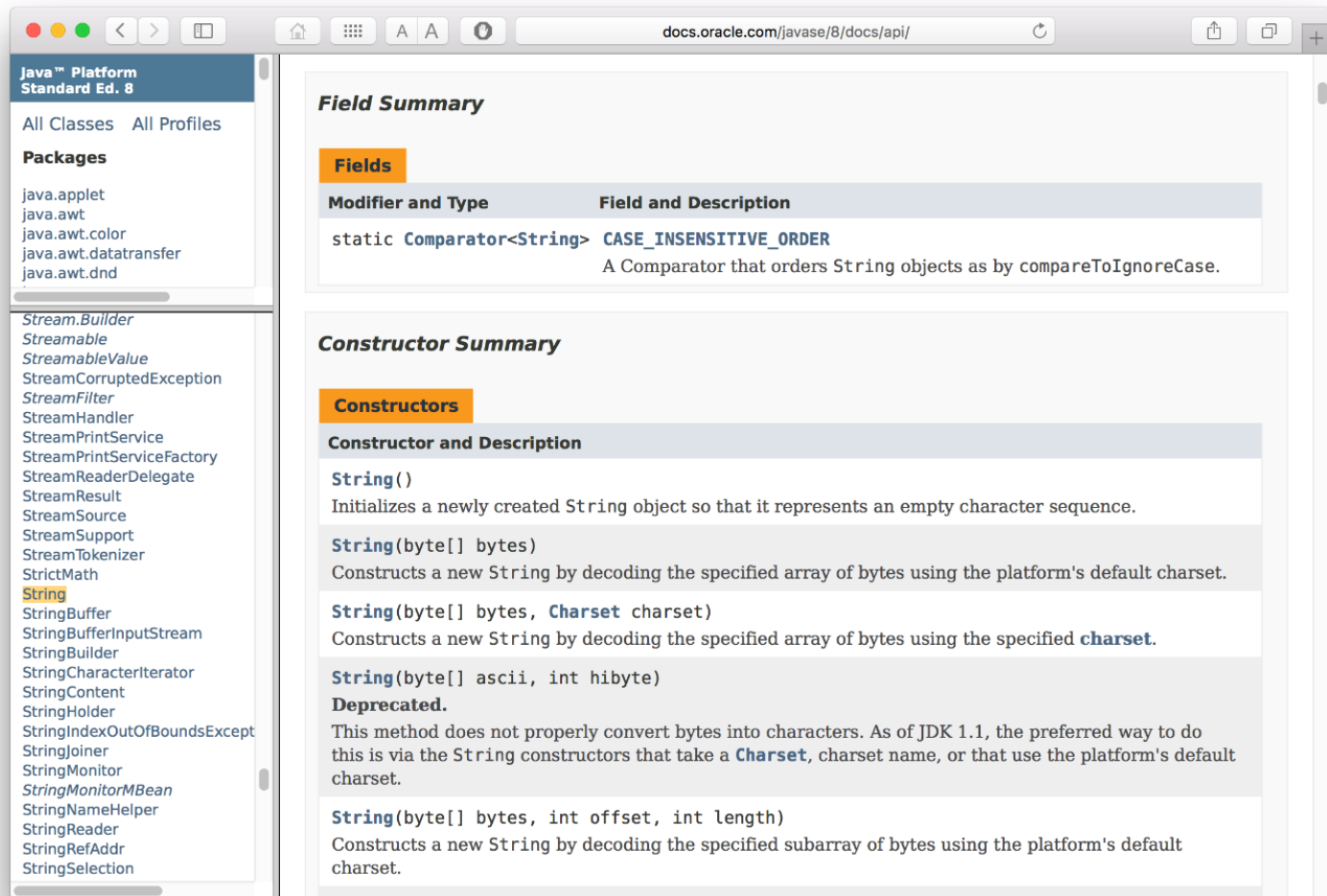
Working with the library

- A competent Java programmer must be able to work with the libraries
- You should:
 - know some important classes by name;
 - know how to find out about other classes
- Remember:
 - we only need to know the *interface*, not the *implementation*

Reading class documentation

- Documentation of the Java libraries in HTML format
- Readable in a web browser
- Class API:
Application Programmers Interface
- Interface description for all library classes

API Reference



The screenshot shows the Java Platform Standard Ed. 8 API Reference website. The left sidebar contains a navigation menu with 'All Classes' and 'All Profiles' at the top, followed by 'Packages' and a list of Java packages including java.applet, java.awt, java.awt.color, java.awt.datatransfer, and java.awt.dnd. Below these is a list of classes, with 'String' highlighted. The main content area is titled 'Field Summary' and contains a section for 'Fields'. This section includes a table with two columns: 'Modifier and Type' and 'Field and Description'. The table lists the static field 'CASE_INSENSITIVE_ORDER' of type 'Comparator<String>'. Below the 'Field Summary' section is the 'Constructor Summary' section, which lists several constructors for the 'String' class, including 'String()', 'String(byte[] bytes)', 'String(byte[] bytes, Charset charset)', 'String(byte[] ascii, int hibyte)', 'String(byte[] bytes, int offset, int length)', and a 'Deprecated' section for 'String(int hibyte)'. The 'Deprecated' section explains that this method does not properly convert bytes into characters and provides a recommendation to use the 'String' constructors that take a 'Charset'.

Java™ Platform
Standard Ed. 8

All Classes All Profiles

Packages

- java.applet
- java.awt
- java.awt.color
- java.awt.datatransfer
- java.awt.dnd

Stream.Builder
Streamable
StreamableValue
StreamCorruptedException
StreamFilter
StreamHandler
StreamPrintService
StreamPrintServiceFactory
StreamReaderDelegate
StreamResult
StreamSource
StreamSupport
StreamTokenizer
StrictMath
String
StringBuffer
StringBufferInputStream
StringBuilder
StringCharacterIterator
StringContent
StringHolder
StringIndexOutOfBoundsException
StringJoiner
StringMonitor
StringMonitorMBean
StringNameHelper
StringReader
StringRefAddr
StringSelection

Field Summary

Fields

Modifier and Type	Field and Description
static Comparator<String>	CASE_INSENSITIVE_ORDER A Comparator that orders String objects as by compareToIgnoreCase.

Constructor Summary

Constructors

Constructor and Description
String() Initializes a newly created String object so that it represents an empty character sequence.
String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
String(byte[] bytes, Charset charset) Constructs a new String by decoding the specified array of bytes using the specified charset .
String(byte[] ascii, int hibyte) Deprecated. This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset , charset name, or that use the platform's default charset.
String(byte[] bytes, int offset, int length) Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

Interface vs implementation

The documentation includes (the WHAT):

- the name of the class
- a general description of the class
- a list of constructors and methods
- return values and parameters for constructors and methods
- a description of the purpose of each constructor and method



the *interface* of the class

Interface vs implementation

The documentation does not include (HOW):

- private fields (most fields are private)
- private methods
- the bodies (source code) of methods

 *the implementation of the class*

Debug with Logging

- Print statements require removal before production
- Logging provides a more effective alternative
 - Import classes/interfaces with core logging facilitate

```
import java.util.logging.Logger;  
import java.util.logging.Level;
```
 - Declare and initialize the logger object for a class

```
Logger logger =  
    Logger.getLogger(ClassName.class.getName());
```
 - Instead of `System.out.println` statements, now use:

```
void log(Level level, String msg)  
logger.log(Level.WARNING, "Name is INVALID");
```
 - Static *Level* constants in descending order:
 - SEVERE (highest value)
 - WARNING
 - INFO
 - CONFIG
 - FINE
 - FINER
 - FINEST (lowest value)

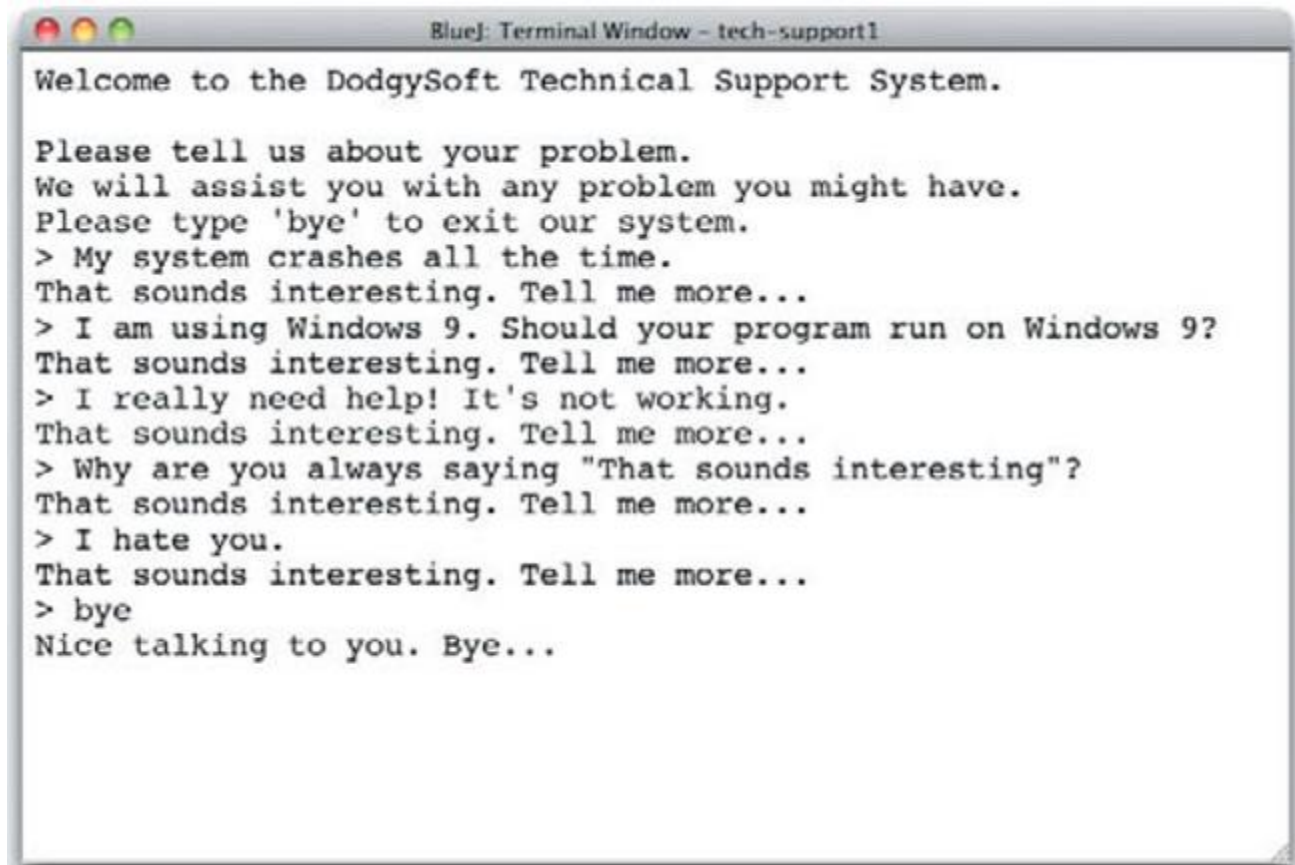


A Technical Support System

- A interactive text dialog system providing technical support for online customers
- Idea based on *Eliza* (AI program) developed by Joseph Weizenbaum (MIT, 1960s)
- *tech-support* project requirements:
 - Technical support system text interface
 - User inputs a technical support question
 - Program generates a response

Interactive text dialog system

tech-support

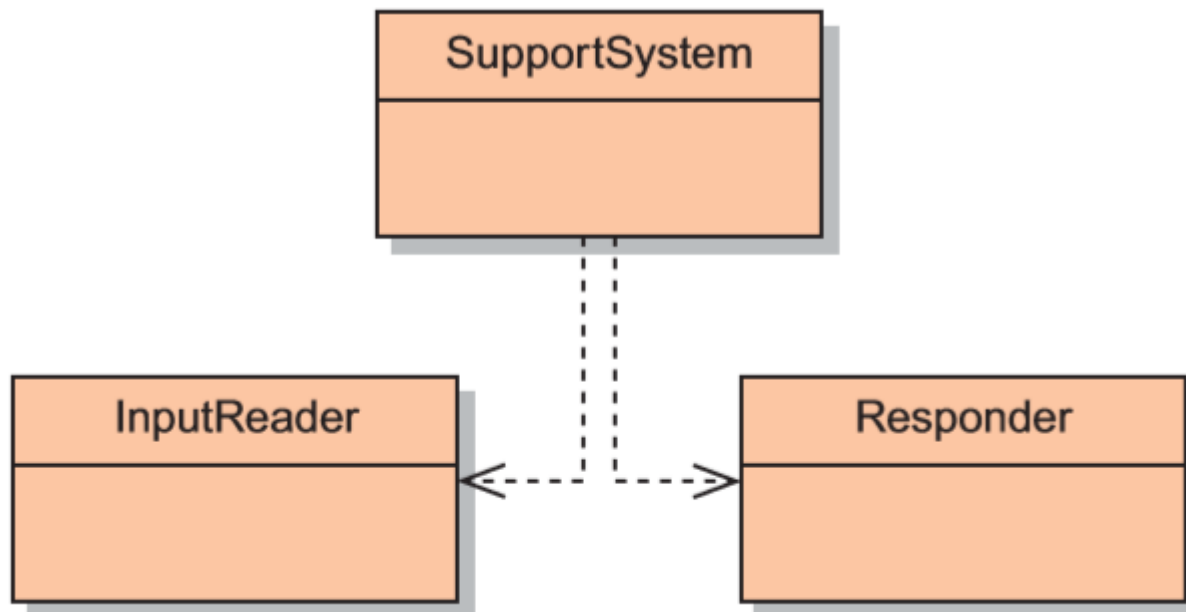


```
BlueJ: Terminal Window - tech-support1

Welcome to the DodgySoft Technical Support System.

Please tell us about your problem.
We will assist you with any problem you might have.
Please type 'bye' to exit our system.
> My system crashes all the time.
That sounds interesting. Tell me more...
> I am using Windows 9. Should your program run on Windows 9?
That sounds interesting. Tell me more...
> I really need help! It's not working.
That sounds interesting. Tell me more...
> Why are you always saying "That sounds interesting"?
That sounds interesting. Tell me more...
> I hate you.
That sounds interesting. Tell me more...
> bye
Nice talking to you. Bye...
```

Modularization



tech-support project

```
public class SupportSystem
{
    private InputReader reader;
    private Responder responder;

    /**
     * Creates a technical support system.
     */
    public SupportSystem()
    {
        reader = new InputReader();
        responder = new Responder();
    }

    ...
}
```

Basic structure

- *tech-support* project requirements:
 - Technical support system text interface
 - User inputs a technical support question
 - Program generates a response

```
String input = reader.getInput();  
...  
String response = responder.generateResponse();  
System.out.println(response);
```


Main loop structure

```
boolean finished = false;
```

```
while(!finished) {
```

```
    do something
```

```
    if(exit condition) {
```

```
        finished = true;
```

```
    }
```

```
    else {
```

```
        do something more
```

```
    }
```

```
}
```

**A common
iteration
pattern.**

SupportSystem class

```
public void start()
{
    boolean finished = false;

    printWelcome();
    while(!finished) {
        String input = reader.getInput();

        if(input.startsWith("bye")) {
            finished = true;
        }
        else {
            String response = responder.generateResponse();
            System.out.println(response);
        }
    }
    printGoodbye();
}
```

**** *input* is ignored by *Responder* in this version**

The exit condition

```
String input = reader.getInput();
```

```
if(input.startsWith("bye")) {  
    finished = true;  
}
```

- Where does 'startsWith' come from?
- What is it?
- What does it do?
- How can we find out?

Documentation for `startsWith`

- `startsWith`
 - `public boolean startsWith(String prefix)`
- Tests if this string starts with the specified prefix
- Parameters:
 - `prefix` - the prefix
- Returns:
 - `true` - if the String startsWith the prefix
 - `false` - otherwise the String does not

Methods from `String`

- `boolean contains(char c)`
- `boolean endsWith(String s)`
- `int indexOf(String s)`
- `int indexOf(String s, int i)`
- `String substring(int b)`
- `String substring(int b, int e)`
- `String toUpperCase()`
- `String trim()`

➤ Beware: strings are *immutable*!

(when invoking its methods, but may be mutated with the assignment operator)

Immutable String

String method

`String toUpperCase()`

Incorrect use

`input.toUpperCase();`

Correct use

`input = input.toUpperCase();`
`if (input.toUpperCase().contains())`

Using library classes

- Classes organized into packages
- Classes from the library must be *imported* using an `import` statement (except classes from `java.lang`)
- They can then be used like classes from the current project

Packages and import

- Single classes may be imported:

```
import java.util.ArrayList;
```

- Whole packages can be imported:

```
import java.util.*;
```

- Importation does NOT involve source code insertion

Using Random

- The library class **Random** can be used to generate random numbers

```
import java.util.Random;  
...  
Random rand = new Random();  
...  
int num = rand.nextInt();  
int value = 1 + rand.nextInt(100);  
int index = rand.nextInt(list.size());
```

Selecting a random response

```
private Random randomGenerator;  
private ArrayList<String> responses;  
  
public Responder()  
{  
    randomGenerator = new Random();  
    responses = new ArrayList<String>();  
    fillResponses();  
}  
  
private void fillResponses()  
{  
    ...fill ArrayList responses with a selection of response strings...  
}  
  
public String generateResponse()  
{  
    int index = randomGenerator.nextInt(responses.size());  
    return responses.get(index);  
}
```

Fill *ArrayList* of responses

```
private ArrayList<String> responses;
```

```
public Responder()
```

```
{
```

```
    randomGenerator = new Random();
```

```
    responses = new ArrayList<String>();
```

```
    fillResponses();
```

```
}
```

```
public void fillResponses()
```

```
{
```

```
    responses.add("That sounds odd. Could you describe \n" + "that problem in more detail?");
```

```
    responses.add("No other customer has ever \n" + "complained about this before. \n" +  
        "What is your system configuration?");
```

```
    responses.add("That's a known problem with Vista." + "Windows 7 is much better.");
```

```
    responses.add("I need a bit more information on that.");
```

```
    responses.add("Have you checked that you do not \n" + "have a dll conflict?");
```

```
    responses.add("That is explained in the manual. \n" + "Have you read the manual?");
```

```
    responses.add("Your description is a bit \n" + "wishy-washy. Have you got an expert \n" +  
        "there with you who could describe \n" + "this more precisely?");
```

```
    responses.add("That's not a bug, it's a feature!");
```

```
    responses.add("Could you elaborate on that?");
```

```
}
```

Parameterized (Generic) classes

- The documentation includes provision for a *type parameter*:
 - `ArrayList<E>`
- These type names reappear in the parameters and return types:
 - `E get(int index)`
 - `boolean add(E e)`

Parameterized classes

- The types in the documentation are placeholders for the types we use in practice
- An `ArrayList<TicketMachine>` actually has methods:
 - `TicketMachine get(int index)`
 - `boolean add(TicketMachine e)`

Review

- Java has an extensive class library
- A good programmer must be familiar with the library
- The documentation tells us what we need to know to use a class (its interface)
- Some classes are parameterized with additional types
 - Parameterized classes are also known as *generic classes* or *generic types*



Further library classes

Using library classes to implement
some more advanced functionality



Main concepts to be covered

- Further library classes:
 - **Set** – avoiding duplicates
 - **Map** – creating associations
- Writing documentation:
 - **javadoc**

Using sets

```
import java.util.HashSet;
```

```
...
```

```
HashSet<String> mySet = new HashSet<String>();
```

```
mySet.add("one");
```

```
mySet.add("two");
```

```
mySet.add("three");
```

```
for(String element : mySet) {  
    do something with element  
}
```

Compare with
code for an
ArrayList!

ArrayList vs. HashSet

- **Similarities**

- Contain a collection of objects
- Add objects (.add method)
- Remove objects (.remove method)
- Number of elements (.size method)
- Iterator ability (.iterator method)

- **Differences**

- HashSet objects are unique, while an ArrayList can have duplicate objects
- HashSet objects are not ordered, while ArrayList objects are ordered

Maps

- Maps are flexible-sized collections that contain:
 - value pairs each with its own object type
 - each pair consists of a key and a value
- Uses the key to easily lookup the value
 - instead of using an integer index
- For example, a telephone book:
 - name and phone number pair
- Reverse-lookup of key using value
 - not so easy

Using maps

- A map with strings as keys and values

:HashMap

"Charles Nguyen"	"(531) 9392 4587"
"Lisa Jones"	"(402) 4536 4674"
"William H. Smith"	"(998) 5488 0123"

Using maps *.put* and *.get*

Declaration and creation of *contacts* HashMap:

```
HashMap<String, String> contacts =  
    new HashMap<String, String>();
```

HashMap *.put* method inserts an entry:

```
contacts.put("Charles Nguyen", "(531) 9392 4587");  
contacts.put("Lisa Jones", "(402) 4536 4674");  
contacts.put("William H. Smith", "(998) 5488 0123");
```

HashMap *.get* method retrieves the value :

```
String number = contacts.get("Lisa Jones");  
System.out.println(number);
```

Does *HashMap* have an *iterator* method?

Using maps in *TechSupport*

```
private HashMap <String, String> responseMap;  
:  
responseMap = new HashMap<String, String>();  
:  
responseMap.put("slow",  
    "I think this has to do with your hardware. \n" +  
    "Upgrading your processor should solve all " +  
    "performance problems. \n" + "Have you got a problem  
    with our software?");  
responseMap.put("bug",  
    "Well, you know, all software has some bugs. \n" +  
    "But our software engineers are working very " +  
    "hard to fix them. \n" + "Can you describe the  
    problem a bit further?");  
responseMap.put("expensive",  
    "The cost of our product is quite competitive. \n" +  
    "Have you looked around and " + "really compared our  
    features?");
```

TechSupport response map

Responses in an ArrayList of String:

```
public String generateResponse()  
{  
    int index = randomGenerator.nextInt(responses.size());  
    return responses.get(index);  
}
```

Using a HashMap:

```
public String generateResponse(String word)  
{  
    String response = responseMap.get(word);  
  
    if(response != null) {  
        return response;  
    }  
    else {  
        return pickDefaultResponse();  
    }  
}
```

Dividing Strings

```
public HashSet<String> getInput()  
{  
    System.out.print("> ");  
    String inputLine =  
        reader.nextLine().trim().toLowerCase();  
  
    String[] wordArray = inputLine.split(" ");  
    HashSet<String> words = new HashSet<String>();  
  
    for(String word : wordArray) {  
        words.add(word);  
    }  
    return words;  
}
```


String *.split*

String[] *split*(String regex)

Splits this string around matches of the given regular expression

```
String[] wordArray = inputLine.split(" ");
```

Splits *inputLine* around the regular expression of “ ”

Regular Expressions

“ ” - space

“\t” - tab

“\\s” - any white space

“[\t]” - space or tab(**grouping**)

“[\t]**+**” - space or tab(**one or more**)

Using *regex*

String.split()

```
String[] wordArray =  
originalString.split("[ \\t]+");
```

Splits original String around (one or more) spaces or tabs

```
String[] wordArray =  
originalString.split("\\s+");
```

Splits original String around (one or more) of ANY white space

String.trim().replaceAll()

```
String newString =  
oldString.trim().replaceAll("\\s+", " ");
```

Removes ALL occurrences of leading and trailing spaces with .trim
AND

Replaces ALL (one or more) white spaces with just a SINGLE space

TechSupport input set

Input using an String:

```
String input = reader.getInput();  
:  
String response = responder.generateResponse();
```

Split input words into a HashSet of String:

```
HashSet<String> input = reader.getInput();  
:  
String response = responder.generateResponse(input);  
:  
public String generateResponse(HashSet<String> words)  
{  
    for(String word : words) {  
        String response = responseMap.get(word);  
        if(response != null) {  
            return response;  
        }  
    }  
    return pickDefaultResponse();  
}
```

List, Map and Set

- Alternative ways to group objects
- Varying implementations available:
 - List: ArrayList, LinkedList
 - Set: HashSet, TreeSet
- HashMap is unrelated to HashSet, & HashSet is closer to ArrayList
- Name consist of 2 parts “Array” “List”
 - 2nd word - collection type (List, Map, Set)
 - 1st word - how it is implemented



Collections and primitive types

- Generic collection classes can be used with all class/object types
- But what about *primitive types* such as `int`, `boolean`, etc...
- Suppose we want an **`ArrayList`** of `int`?

Wrapper classes

- Primitive types are not objects types
- Primitive-type values must be wrapped in objects to be stored in a collection!
- Wrapper classes exist for all primitive types:

<i>Primitive type</i>	<i>Wrapper class</i>
int	Integer
float	Float
char	Character
...	...

Wrapper classes

wrap the value

```
int i = 18;
```

```
Integer iwrap = new Integer(i);
```

```
...
```

```
int value = iwrap.intValue();
```

unwrap it

In practice, *autoboxing* and *unboxing* mean we don't often have to do this explicitly

```
int i = 18;
```

```
Integer iwrap = i;
```

```
...
```

```
int value = iwrap;
```

Autoboxing and unboxing

```
private ArrayList<Integer> markList;  
...  
public void storeMark(int mark)  
{  
    markList.add(mark);  
}
```

autoboxing

```
int firstMark = markList.remove(0);
```

unboxing



Class variables and Constants

Class variables

- A class variable is shared between ALL instances/objects of the class
- It is a field stored in the class and exists independent of any instances
- Designated by the **static** keyword
- Public static variables are accessed via the class name (NOT object name)
 - **Thermometer.boilingPoint**

Constants

- A variable, once set, can have its value fixed
- Designated by the **final** keyword
 - `final int SIZE = 10;`
- Final *fields* must be set in their declaration or the constructor
- Combining **static** and **final** is common

Class constants

- **static** - class variable

- **final** - constant

```
private static final int GRAVITY = 3;
```

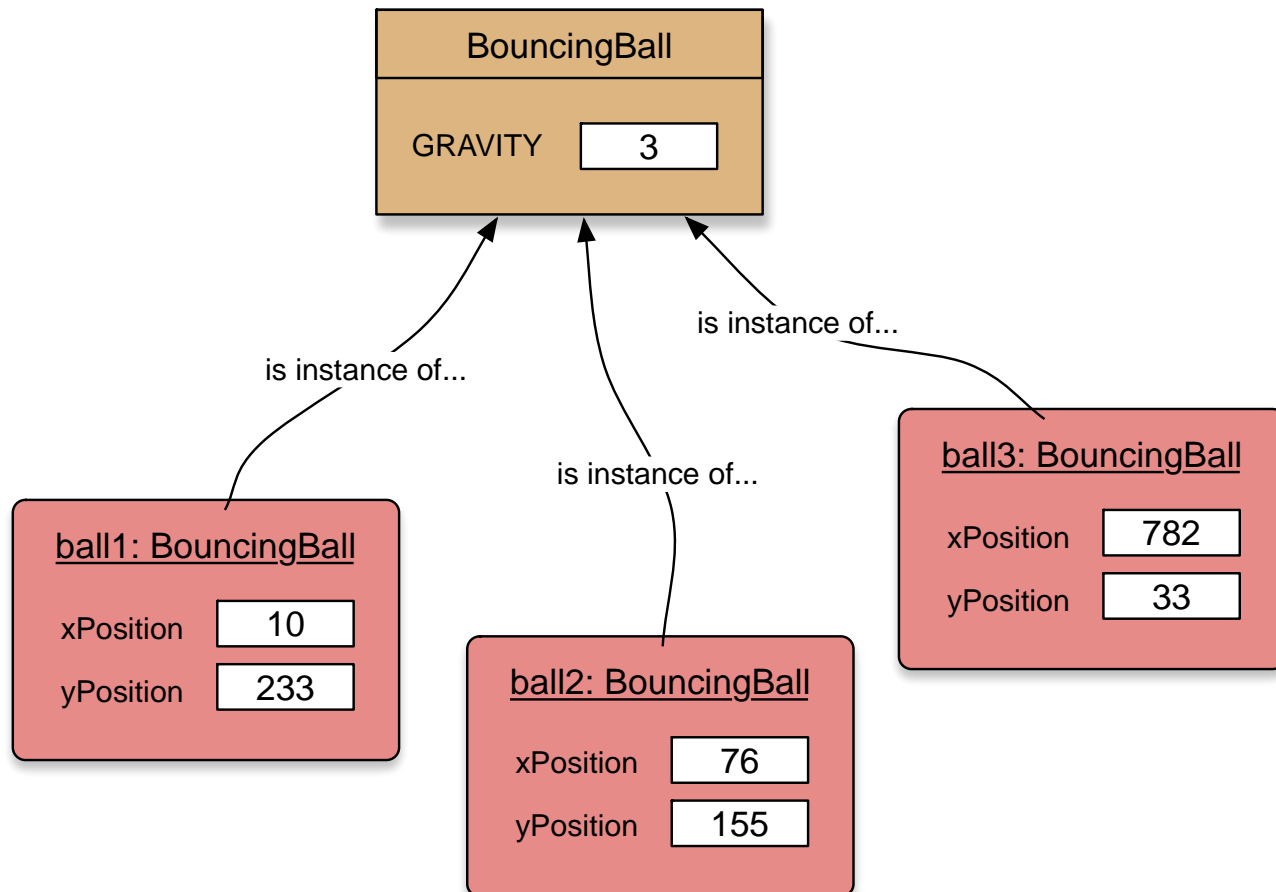
- Public visibility is less of an issue with **final** fields

- Upper-case names often used for class constants:

```
public static final int BOILING_POINT = 100;
```

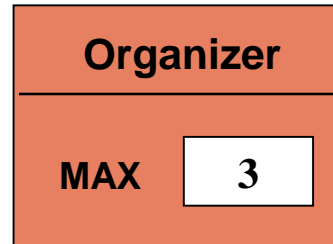

Class constants

```
private static final int GRAVITY = 3;  
private int xPositon;  
private int yPositon;
```

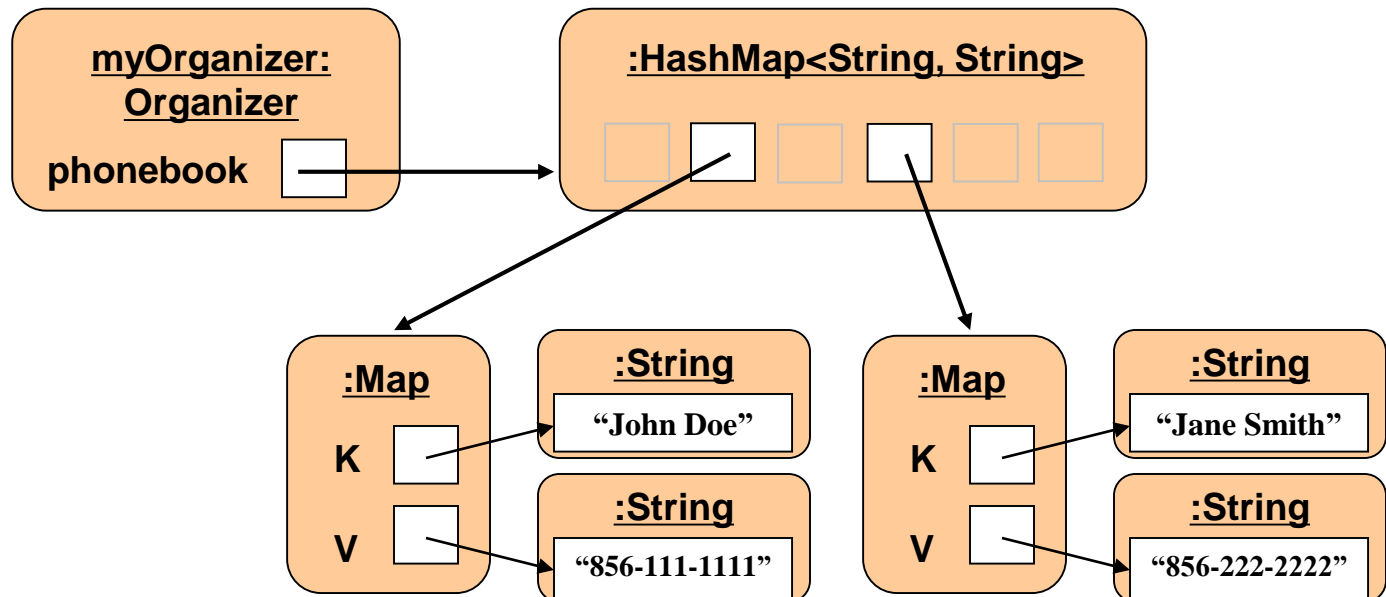


HashMap diagrams

```
private static final int MAX = 3;
```



```
private HashMap<String, String> phonebook;
```



Class Methods

- So far, only used *instance methods*
 - invoked on an instance(object) of a class
- However, *class methods* are different
 - may be invoked WITHOUT a class object
- Similar to class variables in that the class methods BELONG to the class
 - having the class is enough to invoke it

Class methods

- A **static** method belongs to its class rather than the instances:

```
public static int getDaysThisMonth()
```

- Static methods are invoked via their class name:

```
int days = Calendar.getDaysThisMonth();
```

There is NO object so the name of class is used before the dot!



Limitations of class methods

- A static method exists independent of any instances.
- Therefore:
 - They cannot access instance fields within their class
 - They cannot call instance methods within their class



Review

- Class variables belong to their class rather than its instances
- Class methods belong to their class rather than its instances
- Class variables are used to share data among instances
- Class methods are prohibited from accessing instance variables and methods



Review

- The values of **final** variables are fixed
- They must be assigned at declaration or in the constructor (for fields)
- **final** and **static** are unrelated concepts, but they are often used together



Further Advanced Material



Polymorphic collection types

- Different collection classes offer similar interfaces:
 - **ArrayList** and **LinkedList**
 - **HashSet** and **TreeSet**
- Types exist which capture those similarities:
 - **List**
 - **Set**

Polymorphic collection types

- *Polymorphism* allows us to ignore the more specific type in most cases
- Create objects of the *specific* type
- BUT declare variables of the more *general* type:

```
List<Track> tracks = new LinkedList<>();
```

```
Map<String, String> responseMap =  
    new HashMap<>();
```



The Stream `collect` method

- Used to create a new collection object at the end of a pipeline
- The `collect` method takes a `Collector` parameter that accumulates elements of the stream
- We often use the polymorphic collection types for the result

Collecting a filtered stream

```
public List<Sighting> getSightingsOf(String animal)
{
    return sightings
        .stream()
        .filter(record -> animal.equals(
            record.getAnimal()))
        .collect(Collectors.toList());
}
```

java.util.stream.Collectors

static toList method returns
a generic Collector object



Writing class documentation

- User classes should be documented the same way library classes are
- Others should be able to use your class without reading the implementation
- Make your class a potential library class



Elements of documentation

Documentation for a class should include:

- the class name
- a comment describing the overall purpose and characteristics of the class
- a version number (@version)
- the authors' names (@author)
- documentation for each constructor and each method



Elements of documentation

The documentation for each constructor and method should include:

- the name of the method
- the return type (@return)
- the parameter names and types (@param)
- a description of the purpose and function of the method
- a description of each parameter
- a description of the value returned

javadoc

Class comment:

/**

*** The Responder class represents a response
* generator object. It is used to generate an
* automatic response.**

*** @author Michael Kölling and David J. Barnes**

*** @version 1.0 (2011.07.31)**

***/**

javadoc

Method comment:

```
/**
 * Read a line of text from standard input (the text
 * terminal), and return it as a set of words.
 *
 * @param prompt A prompt to print to screen.
 * @return A set of Strings, where each String is
 *         of the words typed by the user
 */
public HashSet<String> getInput(String prompt)
{
    ...
}
```

Public vs private

- **Public** elements are accessible to objects of other classes
 - Fields, constructors and methods
- Fields should NOT be public
 - Keeps the integrity of the field
- **Private** elements are accessible only to objects of the same class
- Only methods that are intended for other classes should be public

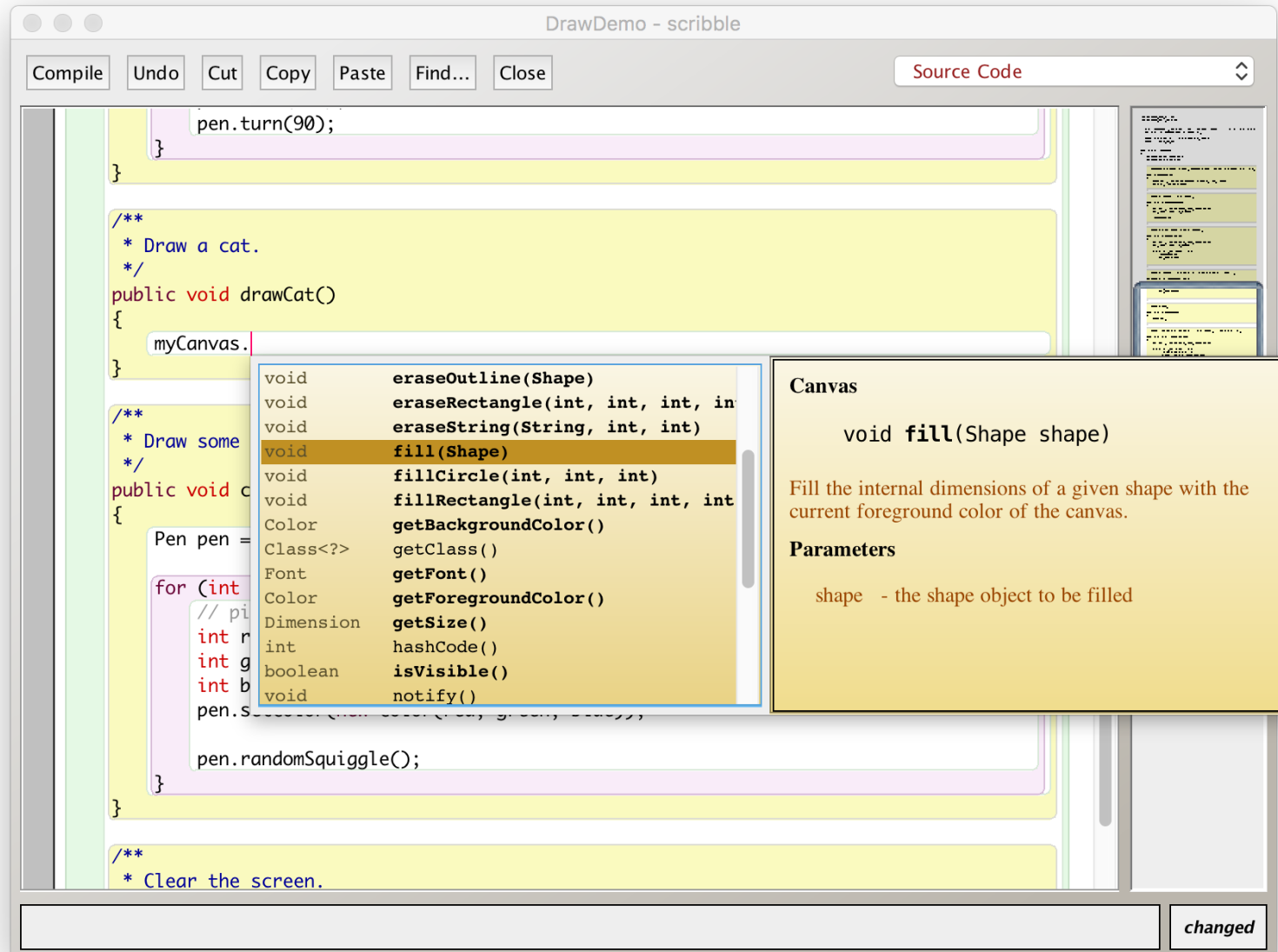
Information hiding

- Data belonging to one object is hidden from other objects (**abstraction**)
- Know what an object can do, but NOT necessarily how it does it
- Information hiding increases the level of *independence* (**modularization**)
- Independence of modules is important for large systems and maintenance (**loose coupling**)

Code completion

- The BlueJ editor supports lookup of methods
- Use **Ctrl-space** after a method-call dot to bring up a list of available methods
- Use *Return* to select a highlighted method

Code completion in BlueJ



Review

- Java has an extensive class library
- A good programmer must be familiar with the library
- The documentation tells us what we need to know to use a class (interface)
- The implementation is hidden (information hiding)
- Classes are documented so that the interface can be read on its own (class comment, method comments)