Programación Orientada a Objetos IV - Uso de colecciones II (sets, maps)

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025

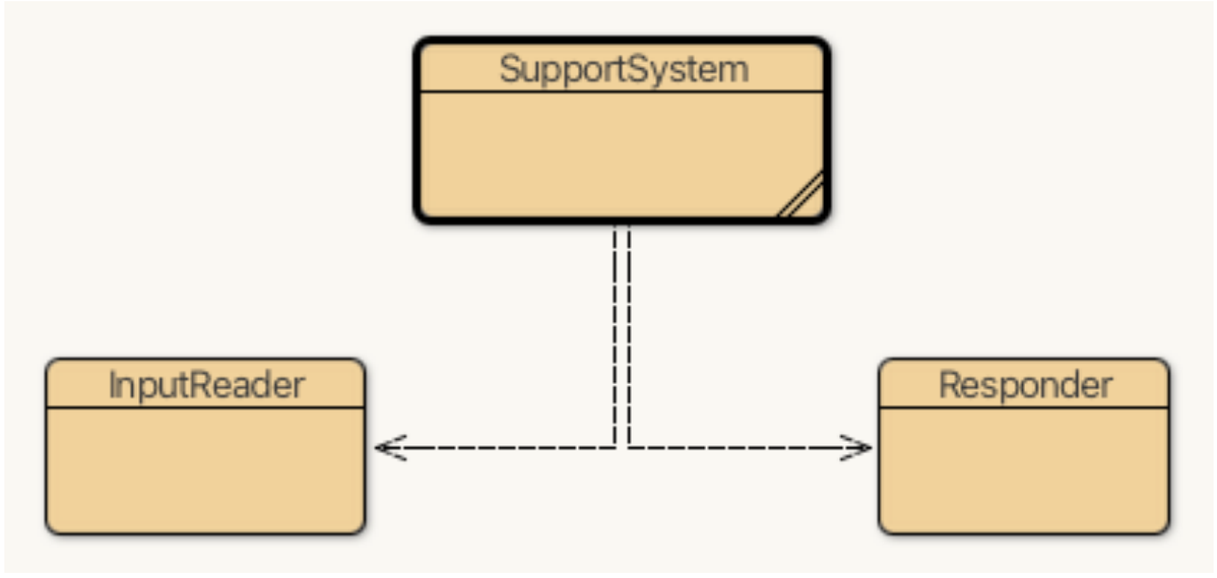
Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

Ejemplo: TechSupport

- TechSupport es un programa para proveer soporte a los clientes de una compañía ficticia llamada DodgySoft
- El propósito del sistema es imitar las respuestas que una persona de soporte podría dar por chat
- La idea de TechSupport está basada en ELIZA, uno de los primeros chatbots
  - Creado en los años 60 por Joseph Weizenbaum, un investigador del MIT en Inteligencia Artificial, para explorar la comunicación entre máquinas y humanos
  - Eliza usaba técnicas pattern matching y sustitución para crear una ilusión de que el programa entendía al usuario
  - En contextos limitados, los humanos con los que interactuaba ELIZA creían que estaban chateando con una persona

TechSupport v1

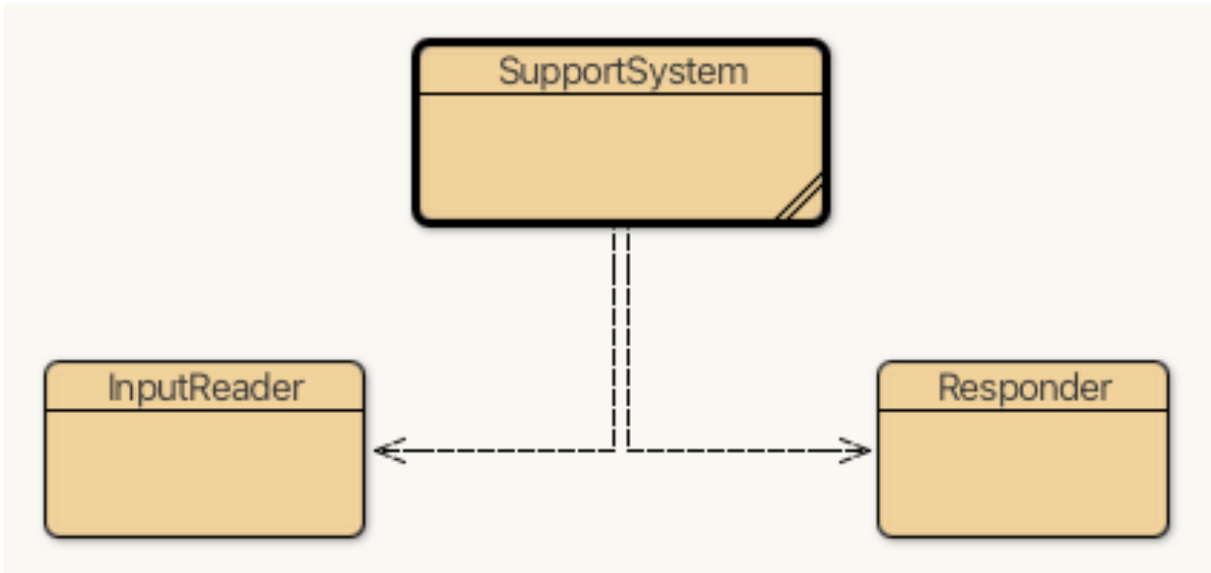
- La primera versión del software TechSupport genera siempre la misma respuesta, independientemente del input del usuario
  - Iremos mejorando el sistema iterativamente
- El sistema consta de 3 clases:
  - `InputReader`: Lee el input del usuario desde la terminal
  - `Responder`: Genera una respuesta fija
  - `SupportSystem`: Implementa la interacción con el usuario. Usa `InputReader` para obtener el input y `Responder` para generar una respuesta



- Notar como cada clase tiene una responsabilidad bien definida
  - Si queremos hacer cualquier cambio en el sistema es fácil saber qué clase debemos modificar (ej. `Responder` si queremos modificar las respuestas)

TechSupport v1

- Uno de los principios de diseño orientado a objetos más importante es el principio de responsabilidad única [1]: Cada clase debe tener una única responsabilidad
  - Los métodos también deben cumplir con este principio
- Explicar la responsabilidad de la clase debería ser sencillo, unas pocas oraciones deberían alcanzar
- En este curso, describimos la responsabilidad de las clases en el comentario antes del encabezado de la clase



- Escribir el comentario nos ayuda a definir de manera precisa la responsabilidad de cada clase, y a adherir al principio de responsabilidad única
- Si una clase tiene más de una responsabilidad, usualmente hay que refactorizar el código para dividirla en dos o más clases, y así lograr un mejor diseño

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Interfaz vs. implementación

- La interfaz de una clase describe qué hace una clase, y como puede usarse, sin mostrar la implementación
- En la mayoría de los casos, conocer la interfaz debería ser suficiente para usar la clase
- La interfaz tiene:
  - El nombre de la clase
  - Una descripción general de su propósito
  - Una lista de los constructores y métodos de la clase
  - Una descripción del propósito de cada constructor y método
- Si se usa un formato predefinido para especificar las clases, una descripción de la interfaz de la clase se puede generar automáticamente
  - En este curso usamos una variante del lenguaje Javadoc

InputReader: Interfaz

```
public class InputReader
extends Object
```

InputReader reads typed text input from the standard text terminal. The text typed by a user is returned.

Constructor Details

InputReader

```
public InputReader()
```

Post:

Create a new InputReader that reads text from the text terminal.

Method Details

getInput

```
public String getInput()
```

Post:

Read a line of text from the text terminal, and return it as a String.

InputReader: Interfaz

Responsabilidad de la clase

```
public class InputReader
extends Object
```

InputReader reads typed text input from the standard text terminal. The text typed by a user is returned.

Constructor Details

InputReader

```
public InputReader()
```

Post:

Create a new InputReader that reads text from the text terminal.

Method Details

getInput

```
public String getInput()
```

Post:

Read a line of text from the text terminal, and return it as a String.

Responder: Interfaz

```
public class Responder
extends Object
```

The responder class represents a response generator object. It is used to generate an automatic response to an input string.

Constructor Details

Responder

```
public Responder()
```

Post:

Construct a Responder.

Method Details

generateResponse

```
public String generateResponse()
```

Post:

Generate a fixed response.

Responder: Interfaz

Responsabilidad de la clase

```
public class Responder
extends Object
```

The responder class represents a response generator object. It is used to generate an automatic response to an input string.

Constructor Details

Responder

```
public Responder()
```

Post:

Construct a Responder.

Method Details

generateResponse

```
public String generateResponse()
```

Post:

Generate a fixed response.

SupportSystem: Interfaz

```
public class SupportSystem
extends Object
```

This class implements a technical support system. It is the top level class in this project. The support system communicates via text input/output in the text terminal. This class uses an object of class InputReader to read input from the user, and an object of class Responder to generate responses. It contains a loop that repeatedly reads input and generates output until the users wants to leave.

Constructor Details

SupportSystem

```
public SupportSystem()
```

Post:

Creates a technical support system.

SupportSystem: Interfaz

```
public class SupportSystem
extends Object
```

This class implements a technical support system. It is the top level class in this project. The support system communicates via text input/output in the text terminal. This class uses an object of class InputReader to read input from the user, and an object of class Responder to generate responses. It contains a loop that repeatedly reads input and generates output until the users wants to leave.

Constructor Details

SupportSystem

```
public SupportSyst
```

Post:

Creates a technical supp

Method Details

start

```
public void start()
```

Post:

Start the technical support system. This will print a welcome message and enter into a dialog with the user, until the user ends the dialog.

SupportSystem: Interfaz

Responsabilidad de la clase

```
public class SupportSystem
extends Object
```

This class implements a technical support system. It is the top level class in this project. The support system communicates via text input/output in the text terminal. This class uses an object of class InputReader to read input from the user, and an object of class Responder to generate responses. It contains a loop that repeatedly reads input and generates output until the users wants to leave.

Constructor Details

SupportSystem

```
public SupportSyst
```

Post:

Creates a technical supp

Method Details

start

```
public void start()
```

Post:

Start the technical support system. This will print a welcome message and enter into a dialog with the user, until the user ends the dialog.

Javadoc

- Los comentarios a continuación son interpretados como especificaciones Javadoc:

```
/**
 This is a javadoc comment.
*/
```

- Javadoc define algunos símbolos especiales para dar formato a la especificación:

```
@version
@author
@param
@return
```

- Y muchos otros...

Javadoc: Ejemplos

- Especificación Javadoc en el código del método `add` de una lista [2]:

```
/**
 * Inserts the specified element at the specified position in this list.
 * Shifts the element currently at that position (if any) and any
 * subsequent elements to the right (adds one to their indices).
 *
 * @param index index at which the specified element is to be inserted
 * @param element element to be inserted
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public void add(int index, E element) {
```

- Notar que hay mucha información redundante en esta especificación: la función de index y element es clara por sus nombres, el nombre del método y el resto de la especificación
- Similarmente, que los elementos siguientes al elemento insertado se corren a la derecha es obvio si uno conoce como funcionan las listas

[2] `java.util.LinkedList`

Javadoc: Ejemplos

- Especificación Javadoc en el código de los métodos `getFirst` y `getLast` de una lista:

```
/**
 * Returns the first element in this list.
 *
 * @return the first element in this list
 * @throws NoSuchElementException if this list is empty
 */
public E getFirst() {
```

- Notar la repetición en la descripción del valor de retorno (`@return`) y la descripción del método (postcondición)

Javadoc: Ejemplos

- Especificación Javadoc en el código de los métodos `getFirst` y `getLast` de una lista:

```
/**
 * Returns the first element in this list.
 *
 * @return the first element in
 * @throws NoSuchElementExceptio
 */
public E getFirst() {
```

```
/**
 * Returns the last element in this list.
 *
 * @return the last element in this list
 * @throws NoSuchElementException if this list is empty
 */
public E getLast() {
```

- Notar la repetición en la descripción del valor de retorno (`@return`) y la descripción del método (postcondición)

Javadoc: Ejemplos

- Usualmente, esta información redundante no es necesaria para comprender el funcionamiento de los métodos
- Como ejemplo, veamos la documentación de las listas [3]:

void	**add**(int index, **E** element) Inserts the specified element at the specified position in this list.
E	**getFirst**() Returns the first element in this list.
E	**getLast**() Returns the last element in this list.

- En este caso, una oración que describe la postcondición es suficiente para entender como funcionan los métodos

[3] https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Comentarios redundantes en Javadoc

- Veamos más ejemplos de comentarios redundantes, tomados el libro Clean Code [1]:

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}
```

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Comentarios redundantes en Javadoc

- Veamos más ejemplos de comentarios redundantes, tomados el libro Clean Code [1]:

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
}

/** The day of the month. */
    private int dayOfMonth;
```

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Comentarios redundantes en Javadoc

- Veamos más ejemplos de comentarios redundantes, tomados el libro Clean Code [1]:

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
 }

/** The day of the month. */
    private int dayOfMonth;

/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Comentarios redundantes en Javadoc

- Veamos más ejemplos de comentarios redundantes, tomados el libro Clean Code [1]:

```
/**
 * Default constructor.
 */
protected AnnualDateRule() {
 }

/** The day of the month. */
    private int dayOfMonth;

/**
 * Returns the day of the month.
 *
 * @return the day of the month.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

```
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;

/** The version. */
private String info;
```

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Comentarios redundantes en Javadoc

- Veamos más ejemplos de comentarios redundantes, tomados el libro Clean Code [1]:

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Los comentarios no compensan un código desordenado

- En el libro Clean Code [1], se menciona que muchas veces se escriben comentarios para intentar compensar la mala calidad del código
  - Es común escribir un módulo confuso y desorganizado, y escribir comentarios para intentar explicar la intención del código
- En cambio, es preferible escribir código claro y expresivo, con comentarios breves y claros, que código desorganizado con muchos comentarios
- En lugar de usar mucho tiempo y esfuerzo escribiendo comentarios largos, es preferible dedicar ese tiempo a escribir buen código:
  - Definir buenas abstracciones de datos
  - Elegir nombres representativos para las clases, métodos, parámetros, etc.
  - Modularizar el código, etc...

[1] Robert C. Martin. "Clean Code: A Handbook of Agile Software Craftsmanship"

Nuestro enfoque en la materia

- Hay comentarios que son muy importantes, que sirven como documentación relevante de la clase, y es crucial contar con ellos:
  - Especificación de las responsabilidades de las clases
  - Pre y postcondiciones de métodos
- De esta manera, en la materia usamos una combinación del formato de comentarios propuesto por Liskov, y algunas de las ideas del libro Clean Code (escribir comentarios breves y claros, evitar la duplicación, etc.)
- En particular, usamos un subconjunto del lenguaje Javadoc

TechSupport v1: Implementación

```java
public class InputReader
{
    private Scanner reader;

    /**
     * @post Create a new InputReader that reads text
     * from the text terminal.
     */
    public InputReader()
    {
        reader = new Scanner(System.in);
    }

    /**
     * @post Read a line of text from the text terminal,
     * and return it as a String.
     */
    public String getInput()
    {
        System.out.print("> ");          // print prompt
        String inputLine = reader.nextLine();

        return inputLine;
    }
}
```

TechSupport v1: Implementación

```
public class InputReader
{
    private Scanner reader;

    /**
     * @post Create a new InputReader that
     * from the text terminal.
     */
    public InputReader()
    {
        reader = new Scanner(System.in);
    }

    /**
     * @post Read a line of text from the
     * and return it as a String.
     */
    public String getInput()
    {
        System.out.print("> ");          //
        String inputLine = reader.nextLine

        return inputLine;
    }
}
```

```
public class Responder
{
    /**
     * @post Construct a Responder.
     */
    public Responder()
    {
    }

    /**
     * @post Generate a fixed response.
     */
    public String generateResponse()
    {
        return "That sounds interesting. Tell me more...";
    }
}
```

TechSupport v1: Implementación

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

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

TechSupport v1: Implementación

```
public class SupportSystem
{
    private InputR
    private Respor

    /**
     * @post Creat
     */
    public Support
    {
        reader = r
        responder
    }
```

```
    /**
     * @post Start the technical support system. This will print a
     * welcome message and enter into a dialog with the user, until
     * the user ends the dialog.
     */
    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();
    }
```

TechSupport v1: Implementación

```
public class SupportSystem
{
    private InputR
    private Respor

    /**
     * @post Creat
     */
    public Support
    {
        reader = r
        responder
    }
```

```
    /**
     * @post Start the technical support system. This will print a
     * welcome message and enter into a dialog with the user, until
     * the user
     */
    public void
    {
        boolean

        printWel

        while(!f
            Stri

            if(i

            }
            else

            }
        }

        printGoodbye();
    }
```

```
    /**
     * @post Print a welcome message to the screen.
     */
    private void printWelcome()
    {
        System.out.println("Welcome to the DodgySoft Technical Support System.");
        System.out.println();
        System.out.println("Please tell us about your problem.");
        System.out.println("We will assist you with any problem you might have.");
        System.out.println("Please type 'bye' to exit our system.");
    }
```

```
    /**
     * @post Print a good-bye message to the screen.
     */
    private void printGoodbye()
    {
        System.out.println("Nice talking to you. Bye...");
    }
```

TechSupport v2

- Nuevos requisitos:
  - Mejorar el sistema para hacer que retorne una respuesta aleatoria, tomada de una lista de respuestas predeterminadas
- Idea para la implementación: Vamos a llenar una `ArrayList` con las respuestas predefinidas, elegir un índice de la lista aleatoriamente, y retornar la respuesta guardada en ese índice
- Debido a que tenemos un sistema bien modularizado, para introducir este cambio sólo tenemos que modificar la clase `Responder` (las clases restantes no sufren cambios)

java.util.Random

- La clase `Random` de `java.util` sirve para generar números pseudoaleatorios (secuencias de números que simulan ser aleatorios)

Constructors

Constructor and Description
`Random()` Creates a new random number generator.
`Random(long seed)` Creates a new random number generator using a single `long` seed.

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`int`	**`nextInt()`** Returns the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence.
`int`	**`nextInt(int bound)`** Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

java.util.Random

- La clase `Random` de `java.util` sirve para generar números pseudoaleatorios (secuencias de números que simulan ser aleatorios)

Constructors

Constructor

`Random()`

Creates a new rand

`Random(long seed)`

Creates a new random number generator using a single `long` seed.

Para una misma semilla, la secuencia de números generada será siempre la misma

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`int`	**`nextInt()`** Returns the next pseudorandom, uniformly distributed `int` value from this random number generator's sequence.
`int`	**`nextInt(int bound)`** Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

TechSupport v2

```
public class Responder
{
    private Random randomGenerator;
    private ArrayList<String> responses;

    /**
     * @post Construct a Responder.
     */
    public Responder()
    {
        randomGenerator = new Random();
        responses = new ArrayList<>();
        fillResponses();
    }
```

TechSupport v2

```
public class Responder
{
    private Random randomGenerator;
    private ArrayList<String> responses;

    /**
     * @post Construct a
     */
    public Responder()
    {
        randomGenerator
        responses = new
        fillResponses()
    }
```

```
/**
 * @post Build up a list of default responses from which we can pick one
 * if we don't know what else to say.
 */
private void fillResponses()
{
    responses.add("That sounds odd. Could you describe this in more detail?");
    responses.add("No other customer has ever complained about this \n" +
                  "before. What is your system configuration?");
    responses.add("I need a bit more information on that.");
    responses.add("Have you checked that you do not have a dll conflict?");
    responses.add("That is covered in the manual. Have you read the manual?");
    responses.add("Your description is a bit wishy-washy. Have you got \n" +
                  "an expert there with you who could describe this better?");
    responses.add("That's not a bug, it's a feature!");
    responses.add("Could you elaborate on that?");
    responses.add("Have you tried running the app on your phone?");
    responses.add("I just checked StackOverflow - they don't know either.");
}
```

TechSupport v2

```
/**
 * @post Generate a random response.
 */
public String generateResponse()
{
    // Pick a random number for the index in the default response
    // list. The number will be between 0 (inclusive) and the size
    // of the list (exclusive).
    int index = randomGenerator.nextInt(responses.size());
    return responses.get(index);
}
```

TechSupport v3

- Hasta el momento el input del usuario no afecta de manera alguna la respuesta del sistema
- Nuevos requisitos:
  - Asociar algunas palabras con respuestas predefinidas
  - Si el input del usuario incluye alguna de estas palabras, retornar la respuesta asociada
  - Sino, retornar una respuesta aleatoria (como veníamos haciendo anteriormente)
- Para esto, el programa tiene que poder:
  - Almacenar palabras con sus respuestas asociadas (vamos a usar maps)
  - Dividir el input del usuario en conjuntos de palabras (vamos a usar sets)

Maps

- Un map es una colección que almacena pares de claves y valores, donde cada clave deben tener un único valor asociado
  - Matemáticamente, un map m es una función parcial:
    $m = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\} \ \ (k_1 \neq k_2 \neq \ldots \neq k_n)$
- Los maps permiten asociar claves con valores, y obtener el valor asociado a una clave de manera eficiente
  - Ej.: En una agenda de contactos podemos usar un map para asociar el nombre de una persona (clave) con su teléfono (valor)
  - El map permite luego buscar el teléfono asociado a un nombre dado de manera eficiente (más eficiente que si almacenamos los pares en una lista)
- Los maps se suelen llamar también diccionarios
- Vamos a utilizar `HashMap`: una implementación de maps de la biblioteca `java.util`
- Más adelante estudiaremos distintas formas de implementar maps y sus características de eficiencia

HashMap: Ejemplo

- Definir un map de nombres de personas (`String`) a números telefónicos (`String`) y agregar algunas entradas:

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

- Obtener e imprimir el número de una persona:

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

java.util.HashMap

Class HashMap<K,V>

https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>

Type Parameters:

`K - the type of keys maintained by this map`

`V - the type of mapped values`

boolean	**containsKey**(**Object** key) Returns `true` if this map contains a mapping for the specified key.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
Set<K>	**keySet**() Returns a **Set** view of the keys contained in this map.
V	**put**(**K** key, **V** value) Associates the specified value with the specified key in this map.
V	**remove**(**Object** key) Removes the mapping for the specified key from this map if present.
int	**size**() Returns the number of key-value mappings in this map.

Maps en TechSupport v3

- Vamos a modificar `Responder` para que almacene asociaciones de palabras y respuestas en un `HashMap`

```
public class Responder
{
    // Used to map key words to responses.
    private HashMap<String, String> responseMap;
    // Default responses to use if we don't recognise a word.
    private ArrayList<String> defaultResponses;
    private Random randomGenerator;

    /**
     * @post Construct a Responder.
     */
    public Responder()
    {
        responseMap = new HashMap<>();
        fillResponseMap();
        [...]
```

Maps en TechSupport v3

- Vamos a modificar `Responder` para que almacene asociaciones de palabras y respuestas en un `HashMap`

```
public class Responder
{
    // Used to map key words to responses
    private H
    // Defau
    private A
    private H

    /**
     * @post
     */
    public Re
    {
        respo
        fillR
        [.
```

```
/**
 * @post Enter all the known keywords and their associated responses
 * into our response map.
 */
private void fillResponseMap()
{
    responseMap.put("crash",
                    "Well, it never crashes on our system. It must have something\n" +
                    "to do with your system. Tell me more about your configuration.");
    responseMap.put("crashes",
                    "Well, it never crashes on our system. It must have something\n" +
                    "to do with your system. Tell me more about your configuration.");
    responseMap.put("slow",
                    "I think this has to do with your hardware. Upgrading your processor\n" +
                    "should solve all performance problems. Have you got a problem with\n" +
                    "our software?");
    responseMap.put("performance",
                    "Performance was quite adequate in all our tests. Are you running\n" +
                    "any other processes in the background?");
```

Sets

- Un conjunto (set) es una colección que almacena cada elemento a lo sumo una vez (no admite repetición), y no mantiene un orden específico para los elementos
  - Matemáticamente, un conjunto s se denota como $s = \{e_1, e_2, \ldots, e_n\}\ (e_1 \neq e_2 \neq \ldots \neq e_n)$
  - Son más apropiados que las listas cuando queremos evitar repeticiones de elementos, o consultar eficientemente si un elemento pertenece o no al conjunto
- Vamos a utilizar `HashSet`: una implementación de sets de la biblioteca `java.util`
- Más adelante estudiaremos distintas implementaciones de conjuntos y sus características de eficiencia

java.util.HashSet

Class HashSet<E>

https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractSet<E>
      java.util.HashSet<E>

Type Parameters:

`E - the type of elements maintained by this set`

boolean	**add**(**E** e) Adds the specified element to this set if it is not already present.
boolean	**contains**(**Object** o) Returns `true` if this set contains the specified element.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(**Object** o) Removes the specified element from this set if it is present.
int	**size**() Returns the number of elements in this set (its cardinality).

Sets en TechSupport v3

- Vamos a modificar `InputReader` para que retorne el conjunto de palabras que ingresó el usuario (en lugar del `String` en crudo)

```java
/**
 * @post Read a line of text from the text terminal,
 * and return it as a set of words.
 */
public HashSet<String> getInput()
{
    // print prompt
    System.out.print("> ");
    String inputLine = reader.nextLine();
    // Remove leading and trailing whitespaces
    inputLine = inputLine.trim();
    // split at spaces
    String[] wordArray = inputLine.split(" ");

    // add words from array into hashset
    HashSet<String> words = new HashSet<>();
    for(String word : wordArray) {
        words.add(word.toLowerCase());
    }
    return words;
}
```

TechSupport v3: Responder

- Ahora estamos listos para que `Responder` elija una respuesta en base a un conjunto de palabras:

```
/**
 * @post Generate a response based on the provided 'words'.
 *   If none of the words can be matched to an appropriate
 *   response, generate a random response.
 */
public String generateResponse(HashSet<String> words)
{
    for (String word : words) {
        String response = responseMap.get(word);
        if(response != null) {
            return response;
        }
    }

    // If we get here, none of the words from the input line was recognized.
    // In this case we pick one of our default responses (what we say when
    // we cannot think of anything else to say...)
    return pickDefaultResponse();
}
```

TechSupport v3: SupportSystem

- Finalmente, debemos hacer leves modificaciones al método `start` de `SupportSystem` para que use los nuevos métodos de `InputReader` y `Responder`

```
/**
 * @post Start the technical support system. This will print a
 * welcome message and enter into a dialog with the user, until
 * the user ends the dialog.
 */
public void start()
{
    boolean finished = false;

    printWelcome();

    while(!finished) {
        HashSet<String> input = reader.getInput();

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

TechSupport v4

- Supongamos que la compañía está recibiendo quejas de sus usuarios porque algunas de las respuestas del sistema no tienen relación alguna con las preguntas realizadas
- La compañía podría decidir analizar las palabras más usadas en las preguntas, para agregar respuestas específicas para estas palabras
- Nuevos requisitos:
  - Almacenar la cantidad de veces que se repite cada palabra en las entradas provistas por el usuario

Wrapper classes

- Hemos visto que las colecciones pueden almacenar objetos de tipo arbitrario
- Sin embargo, las colecciones no pueden almacenar tipos primitivos
- Para solucionar este problema, Java provee las denominadas clases wrapper (envoltorio)

Primitive type	Wrapper type
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

- Cada tipo primitivo tiene una clase wrapper correspondiente, que representa el mismo tipo pero como objetos ("envuelve" los valores primitivos en objetos)
- Ejemplo: Si `ix` es una variable de tipo `int`, la envolvemos con:

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

- Las colecciones sólo pueden declararse usando el tipo del wrapper:

```
private ArrayList<Integer> markList;
```

Autoboxing

- Cuando se usa un tipo primitivo en un contexto en el que se requiere un objeto wrapper, el compilador de Java realiza la conversión automáticamente
  - Esta operación se denomina autoboxing
  - Por ejemplo, se puede agregar un `int` a una colección que almacena `Integer`s:

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

- La operación inversa se denomina unboxing, y también la realiza automáticamente por el compilador
  - Por ejemplo, para obtener un `int` de una colección que almacena `Integer`s:

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

TechSupport v4

- Para contar las palabras ingresadas por los usuarios vamos a implementar una clase `WordCount`, que lleve la cuenta usando un map
  - Mapeamos cada palabra (`String`) a la cantidad de veces que se utilizó (`Integer`)
- Notar que esta abstracción es reusable: podría usarse en muchas otras aplicaciones que requieran contar palabras
  - Esa es una de las ventajas de definir una nueva clase en lugar de usar un atributo de tipo map dentro de `SupportSystem`

```java
/**
 * Keep a record of how many times each word was
 * entered by users.
 */
public class WordCounter
{
    // Associate each word with a count.
    private HashMap<String, Integer> counts;

    /**
     * @post Create a WordCounter
     */
    public WordCounter()
    {
        counts = new HashMap<>();
    }

    /**
     * @post Update the usage count of all words in 'input'.
     */
    public void addWords(HashSet<String> input)
    {
        for(String word : input) {
            int counter = counts.getOrDefault(word, 0);
            counts.put(word, counter + 1);
        }
    }
```

TechSupport v4

- Las listas, sets y maps son estructuras de bajo nivel que deberían usarse como parte de la implementación de las clases
- Si queremos abstraernos de la implementación de las clases tenemos evitar pasar como parámetros y retornar estas estructuras tanto como sea posible
- Por ejemplo, definir un método `int getCount(String word)` nos ayuda a abstraernos de la implementación, ya que oculta a los clientes de la clase que estamos usando un map para implementarla
- Con buenas abstracciones favorecemos el reuso, y podemos modificar la implementación de las clases sin afectar a los clientes

```java
/**
 * Keep a record of how many times each word was
 * entered by users.
 */
public class WordCounter
{
    // Associate each word with a count.
    private HashMap<String, Integer> counts;

    /**
     * @post Create a WordCounter
     */
    public WordCounter()
    {
        counts = new HashMap<>();
    }

    /**
     * @post Update the usage count of all words in 'input'.
     */
    public void addWords(HashSet<String> input)
    {
        for(String word : input) {
            int counter = counts.getOrDefault(word, 0);
            counts.put(word, counter + 1);
        }
    }
```

TechSupport v4

- Las listas, sets y maps son estructuras de bajo nivel que deberían usarse como parte de la implementación de las clases
- Si queremos abstraernos de la implementación de las clases tenemos evitar pasar como parámetros y retornar estas estructuras tanto como sea posible
- Por ejemplo, definir un método `int getCount(String word)` nos ayuda a abstraernos de la implementación, ya que oculta a los clientes de la clase que estamos usando un map para implementarla
- Con buenas abstracciones favorecemos el reuso, y podemos modifica clases si

Ejercicio: Implementar la versión 4 de TechSupport

```
/**
 * Keep a record of how many times each word was
 * entered by users.
 */
public class WordCounter
{
    // Associate each word with a count.
    private HashMap<String, Integer> counts;

    /**
     * @post Create a WordCounter
     */
    public WordCounter()
    {
        counts = new HashMap<>();
    }

    /**
     * @post Update the usage count of all words in 'input'.
     */
    public void addWords(HashSet<String> input)
    {
                                                    (word, 0);
        }
    }
```

Resumen: Listas, conjuntos, maps

- En la actualidad, las colecciones están presentes en todo tipo de software
- Algunos de los tipos de colecciones más importantes son:
  - Listas: Representan secuencias ordenadas de elementos $[e_1, e_2, \ldots, e_n]$
    - Las usamos cuando en nuestra aplicación nos importa preservar el orden de los elementos
    - Permiten el acceso eficiente a una posición dada
  - Conjuntos: Representan conjuntos de elementos $\{e_1, e_2, \ldots, e_n\}$
    $(e_1 \neq e_2 \neq \ldots \neq e_n)$
    - Los usamos cuando no nos importa el orden y/o cuando queremos evitar almacenar elementos repetidos
    - Permiten consultar eficientemente sobre la pertenencia de un elemento
  - Maps: Representan funciones parciales $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$
    $(k_1 \neq k_2 \neq \ldots \neq k_n)$
    - Los usamos cuando queremos mantener asociaciones entre claves y valores
    - Permiten obtener eficientemente el valor asociado a una clave

Resumen: Listas, conjuntos, maps

- Cada uno de estos tipos de colecciones admite muchas implementaciones diferentes
  - Y las operaciones de las distintas implementaciones presentan distintas características de eficiencia, que las hacen más (o menos) apropiadas para cada tipo de aplicación particular
- Implementar eficientemente estas colecciones (y otras) es uno de los temas centrales de esta materia, y lo estudiaremos en detalle más adelante

Modificadores de acceso y ocultamiento de información

- Los modificadores de acceso definen la visibilidad de un atributo o método
  - `public` indica que el atributo o método es accesible en la clase que lo define y en cualquier otra clase
  - `private` indica que el atributo o método sólo es accesible en la clase que lo define
- El ocultamiento de información es un principio de diseño que indica que los detalles de implementación de una clase deben estar ocultos para las otras clases
- Este principio favorece una mejor modularización de las aplicaciones
- Si seguimos este principio, en muchos casos vamos a poder cambiar la implementación de una clase sin afectar a sus clientes
  - Por ejemplo, durante el mantenimiento del sistema para:
    - Mejorar la eficiencia,
    - Corregir defectos,
    - Mejorar el código,
    - etc.

Ocultamiento de información

- Para adherir al principio de ocultamiento de información vamos a tener en cuenta las siguientes pautas
- Recordar que la interfaz de una clase define su comportamiento visible para el resto de las clases
- La interfaz es la parte pública de la clase, y está conformada por sus métodos públicos
    - Si queremos hacer que un método sea parte de la funcionalidad provista por la clase, lo definimos como público
- La implementación define como funciona la clase internamente
    - La implementación es la parte privada de la clase
    - Los atributos de las clases son parte de la implementación y deben ser privados (salvo algunas excepciones)
    - Los métodos que proveen funcionalidades internas a la clase deben ser privados

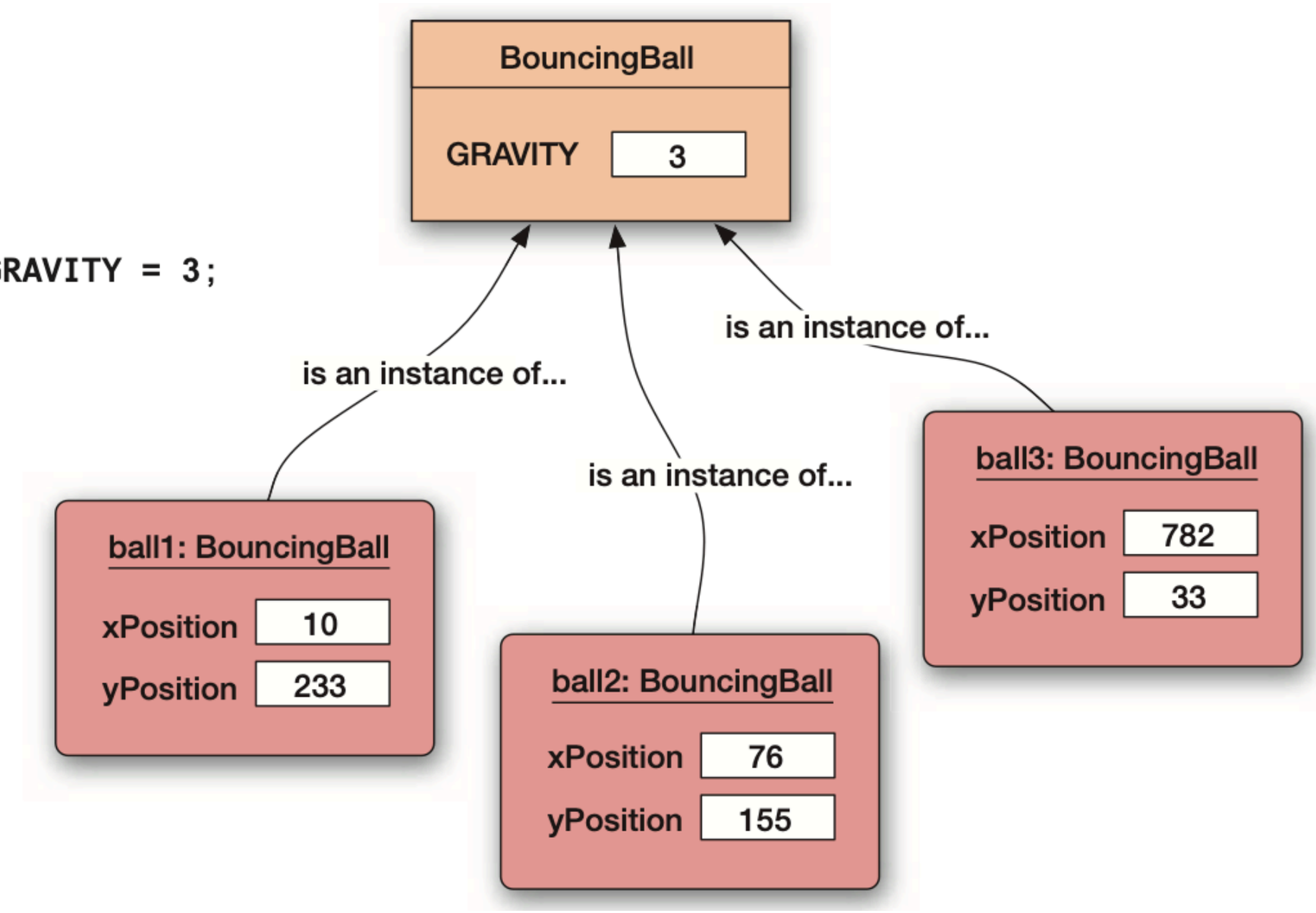
Atributos de clase

- Las clases pueden tener atributos, que se definen con la palabra reservada `static`
  - También se denominan atributos estáticos
- Un atributo de clase tiene una única copia para toda la clase, independientemente de la cantidad de objetos que se creen
- La palabra reservada `final` indica que el atributo de clase es también una constante
  - Las constantes deben ser inicializadas en su declaración
- La definición de constantes es uno de los usos más frecuentes de los atributos de clase en la práctica

```
public class BouncingBall
{
    // Effect of gravity.
    private static final int GRAVITY = 3;
    private int xPosition;
    private int yPosition;
    Other fields and method omitted.
}
```

Atributos de clase vs. atributos

```
public class BouncingBall
{
    // Effect of gravity.
    private static final int GRAVITY = 3;
    private int xPosition;
    private int yPosition;
    Other fields and method omitted.
}
```



Métodos de clase

- Los métodos de clase pertenecen a una clase (en lugar de a sus instancias)
  - También se denominan métodos estáticos
- Se definen dentro de una clase con la palabra reservada `static`

```
public static int getNumberOfDaysThisMonth()
{
    ...
}
```

- Los métodos de clase se invocan usando el nombre de la clase que los define:

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

- Los métodos de clase no pueden acceder a atributos ni invocar a métodos de los objetos
- No ayudan a la definición de buenas abstracciones de datos, por lo que trataremos de no usarlos
  - Salvo en circunstancias muy específicas, como por ejemplo, para definir operaciones sobre tipos primitivos

El método main

- El método `main` es el punto de inicio para la ejecución de las aplicaciones Java
- Debe tener el siguiente perfil:

```
public static void main(String[] args)
```

  - El arreglo de `String`s `args` permite pasar valores como parámetro en la invocación del programa
    - Por ejemplo, desde la terminal
  - `main` debe ser un método de clase (`static`)

- Usualmente se define una clase aparte que contiene este método, para separar el punto de inicio del resto de las clases de la aplicación
  - Por ejemplo, podemos nombrar esta clase como `Main`, o con el nombre del proyecto

El método main

- Como ejemplo, veamos el método `main` para nuestro proyecto `TechSupport`

```
/**
 * Main class for the TechSupport project.
 */
public class Main
{
    public static void main(String[] args) {
        SupportSystem system = new SupportSystem();
        system.start();
    }
}
```

Actividades

- Leer el capítulo 6 del libro "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016.

Bibliografía

- "Objects First with Java A Practical Introduction using BlueJ". Sixth Edition. D. Barnes & M. Kölling. Pearson. 2016.
- "Program Development in Java - Abstraction, Specification, and Object-Oriented Design". B. Liskov & J. Guttag. Addison-Wesley. 2001.
- "Clean Code: A Handbook of Agile Software Craftsmanship (1st. ed.)". Robert C. Martin. Prentice Hall. 2008.