Data structures

Vector

- Expandable and contractible array of objects.
- Empty at creation.
- The initial capacity and how the vector will size itself can be controlled by its **initialCapacity** and **capacityIncrement** fields which can be specified in one of the overloading constructors.
- Examples:
  - `Vector v = new Vector(); // Creates an empty Vector.`
  - `Vector v = new Vector(25); // Creates an empty Vector with initial capacity of 25.`
  - `Vector v = new Vector (25, 5); // Creates an empty Vector with initial capacity of 25 which will grow by 5 each time the Vector overflows.`

Vector operations

- The **add(E e)** method is used to add an element to the Vector.
- The **add(int index, E e)** is used to add an element to the Vector at a specified position.
- The **set(int index, E e)** is used to replace the existing element at the specified position.
- The **get(int index)** method is used to retrieve an element at the specified index.
- The **remove(int index)** method is used to remove the element at the specified index.
- The **size()** method is used to determine the number of elements in the Vector.
- The **isEmpty()** method is used to check if there are no elements in the Vector.
- The **contains(Object o)** is used to check whether an object is stored in the Vector.
- The **indexOf(Object o)** is used to get the index of the object in the Vector.
- The **clear()** method is used to remove all elements from the Vector.
- The **iterator()** method is used to generate an Iterator object which can be used to iterate the Vector.
- The **toArray()** method is used to create an array that contains the elements of the Vector.

Vector operation examples

- `v.add("Doe"); // Adds a new element.`
- `v.add(0, "John"); // Adds the String "John" at index 0.`
- `String s = v.get(1); // s will reference the String "Doe".`
- `v.indexOf("Doe"); // returns 1.`
- `v.contains("Doe"); // returns "true".`
- `v.remove(0); // Removes the element at index 0.`
- `v.clear(); // Clears all elements.`

Iterator

- The Iterator interface provides a standard means of iterating through a collection of elements in a defined sequence.
  - The **hasNext()** method determines whether the structure contains any more elements.
  - The **next()** method retrieves the next element in a structure. If there are no more elements, next() throws a **NoSuchElementException** exception.
  - The **remove()** method removes the last element returned by the next() method from the underlying collection.
- Example:

```
while (users.hasNext()) {
    Object ob = users.next();
    System.out.println(ob);
}
```

Looping through data structures

- Most of the data structures provide an iterator, an implementation of the Iterator interface which can be used to traverse its elements.

  Example:

```
for (Iterator i = v.iterator(); i.hasNext(); ) {
    String name = (String) i.next();
    System.out.println(name);
}
```

- Java provides an enhanced for loop for collections that implement the Iterator interface:

```
for (String name : v) {
    System.out.println(name);
}
```

Example: CodeKeeper

Stacks

- A stack is a data structure used to model information accessed in a specific order. The **Stack** class in Java is implemented as a last-in-first-out (LIFO) stack which means that the last item added to the stack is the first one to be removed.
- The Stack class actually extends the Vector class which means that it supports all operations supported by Vector. It additionally offers several methods for convenience when working with stacks such as:
  - `s.push("One"); // Adds an element on the top of the stack.`
  - `String str = s.pop(); // Assigns the element from the top of the stack to str and removes the element from the stack.`
  - `String str = s.peek(); // Assigns the element from the top of the stack to str but doesn't remove it from the stack.`
  - `int i = s.search("One"); // Returns the distance from the top of the stack of the element if it exists on the stack.`

Map

- **Map** is an interface which defines a number of key-value data structures i.e. a way to store objects referenced by a key. The key serves the purpose of an index in an array or a List such as a Vector - it's a unique value used to access the data stored at a position in the data structure.
- Java provides a number of implementations of the Map interface such as **Hashtable**.
- Example:

```
Hashtable look = new Hashtable();
Rectangle r1 = new Rectangle(0, 0, 5, 5);
look.put(“small”, r1);
Rectangle r3 = new Rectangle(0, 0, 25, 25);
look.put(“large”, r3);
Rectangle r = look.get(“large”);
look.remove(“large”);
```

Example: ComicBooks

The problem with data structures

Data structures generally accept any type of object as their elements. This flexibility can be a good thing since a method that accepts a collection as an argument can be implemented in such way to handle any type of e.g. text representation such as Strings, character arrays, StringBuffers etc. However, if we misplace an Integer inside the data structure, the method will result with an error.

Example:

Instead of

```
quality.put(“near mint”, 1.50F);
```

we code as

```
quality.put(“near mint”, “1.50”);
```

The class compiles successfully but when it's run, it stops with a ClassCastException on the following statement:

```
comix[1].setPrice((Float) quality.get(comix[1].condition));
```

Generics

The solution is to tell the compiler what type of objects are permitted in the data structure using the "<" and ">" characters during the declaration and instantiation of the data structure.

Example:

```
Vector<Integer> zipCodes = new Vector<Integer>;
zipCodes.add(1000); // This works.
zipCodes.add(“1000”); // The statement fails with a compile time error;
```

Example: CodeKeeper2