
# Generic Collections

## Lists

We've already seen and used the generic **List** collection in the Java:

A generic type argument. Means this ArrayList will contain **Strings**.

```
List<String> strList = new ArrayList<String>();
strList.add("Hello");
strList.add("World!");

List<Integer> integerList = new ArrayList<Integer>();
integerList.add(1);
integerList.add(10);

List<User> MyList = new ArrayList<User>();
MyList.add(new User("Nylund"));
MyList.add(new User("Admin"));
```

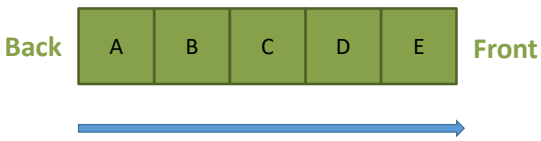
Means this ArrayList will contain **Integers**.

Means this ArrayList will contain **User objects**.

# Queues

## FIFO-queues

The **queue** class is a **First In, First Out** collection (**FIFO**)




- When we **add** items, they are added to the **back**.
- When we **remove** items, they are removed from the **front**.

## Example (1)

In Java, **Queue** is a **interface.** As with **List**, we need to use an **implementation** of Queue.

```
public interface Queue<E> extends Collection<E> {
    boolean add(E e);
    boolean offer(E e);
    E remove();
    E poll();
    E element();
    E peek();
}
```

| Operation | Description |
|---|---|
| add(element) | Adds element to end of queue. Exception if queue full. |
| offer(element) | Adds element to queue. Returns false if queue full. |
| remove() | Removes element from queue. |
| poll() | Removes element at head of queue. Returns null if empty. |
| element() | Looks at element in head of queue. Exception if empty |
| peek() | Looks at element in head of queue. Does not remove. |

## Example (2)

Many classes implements the Queue interface, including:

| Type | Type |
|---|---|
| AbstractQueue | LinkedBlockingQueue |
| ArrayBlockingQueue | **LinkedList** |
| ArrayDeque | LinkedTransferQueue |
| ConcurrentLinkedDeque | PriorityBlockingQueue |
| ConcurrentLinkedQueue | PriorityQueue |
| DelayQueue | SynchronousQueue |
| LinkedBlockingDeque | |

We will use the **LinkedList** implementation in our example

## Example (3)

Adding items to a queue:

```
Queue<String> queue = new LinkedList<String>();
queue.add("A");
queue.add("B");
queue.add("C");

queue.remove();

queue.add("X");
queue.add("Y");
queue.add("Z");
```

## Example (4)

Since this is a **FIFO** queue, the first item to be de-queued should also be the first one we added.

We'll try this with a **while**-loop:

```
while (queue.size() > 0) {
    String current = queue.remove();
    System.out.println(current);
}
```

```
B
C
X
Y
Z
```

## Queues – Removing items

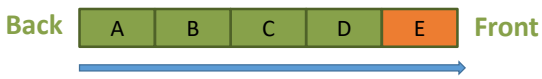
The **remove** method does two things:

```
String current = queue.remove();
```

- Remove the object from the front of the queue.
- Return the object that was removed.
- Throws an exception if queue is empty.

## Queues – Peeking

We might want to just **retrieve** the object at the front of the queue, without actually removing it.




To do so, we can call the **peek** method:

```
// This will NOT remove the element from the queue
String current = queue.peek();
System.out.println(current);
```

Peek() will return null if the queue is empty

# Stacks

## Stack

While a **queue** was a **FIFO** structure, there's also the concept of a **LIFO** structure (**Last In, First Out**).

A **stack** is such a structure.

## Pushing

Adding data to the top of a stack is called **pushing**.

```
Stack<String> stack = new Stack<String>();
stack.push("A");
...
```


We **Push** items to the stack

## Pushing

Removing data from the queue is called **popping**.

```
Stack<String> stack = new Stack<String>();
stack.push("A");
...
String str= stack.pop();
...
```


We **Pop** to remove items from a stack

## Example

The built in **Stack** type provides the following methods besides the ones from its base-class:

| Operation | Description |
|---|---|
| empty() | Test if the stack is empty |
| peek() | Looks at the element at the top of the stack. Does not remove. |
| pop() | Removes the item on the top of the stack |
| push(E item) | Pushes an item onto the top of the stack |
| search(Object O) | Search for an object on the stack |

## Example

We'll use the same example as we did with **queue**, but use a **stack**:

```
Stack<String> stack = new Stack<String>();
stack.push("A");
stack.push("B");
stack.push("C");
stack.pop();
stack.push("X");
stack.push("Y");
stack.push("Z");

while (stack.size() > 0) {
    String current = stack.pop();
    System.out.println(current);
}
```

Considering that stacks are **LIFO** structures, what should the output be?

## Example

Output:

```
Stack<String> stack = new Stack<String>();
stack.push("A");
stack.push("B");
stack.push("C");
stack.pop();
stack.push("X");
stack.push("Y");
stack.push("Z");

while (stack.size() > 0) {
    String current = stack.pop();
    System.out.println(current);
}
```


## Example

You can use the **peek** method on a **stack** as well.

While **pop** removes the object at the top and returns it, **peek** will just return it, leaving the stack as it is.

```
// This will NOT remove the object from the stack
String item = stack.peek();
System.out.println(item);
```

# Maps

## Maps

Let's say we have List of +10000 customers:

```
public class Customer
{
    public int CustomerId;
    public String Name;

    public Customer(int customerId, String name)
    {
        CustomerId=customerId;
        Name=name;
    }
}
```

```
List<Customer> customers = new ArrayList<>();

customers.add(new Customer(1,"Carl"));
customers.add(new Customer(12,"Eric"));
customers.add(new Customer(252,"Tore"));
customers.add(new Customer(1021,"Max"));

//adding 10000's more

customers.add(new Customer(39421,"Fredrik"));
```

This is how the list is organized:

| Index | Customer |
|---|---|
| [0] | {1, Carl} |
| [1] | {12, Eric} |
| … | … |
| [99999] | {39421, Fredrik} |

## Maps

What problems can we have using this list?

| Index | Customer |
|---|---|
| [0] | {1, Carl} |
| [1] | {12, Eric} |
| … | … |
| [99999] | {39421, Fredrik} |

- How do we find customer with ID=95344?
- Check if customer with ID exists?
- Remove customer from list?
- What about performance?

## Maps

A better solution would be to have a structure that look like this instead:

| Index | Customer |
|---|---|
| [0] | {1, Carl} |
| [1] | {12, Eric} |
| … | … |
| [99999] | {39421, Fredrik} |

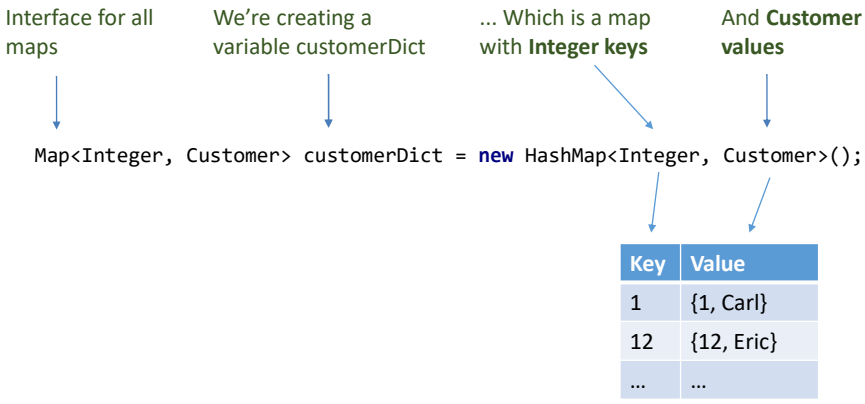
| Key | Value |
|---|---|
| 1 | {1, Carl} |
| 12 | {12, Eric} |
| … | … |
| 39421 | {39421, Fredrik} |

Her we can directly lookup the customer by CustomerID!

Awesome! But how do we implement that?

# Maps

In Java we can implement this using **Map** and **HashMap**




There are several implementations of the Map interface, to see all visit :
http://docs.oracle.com/javase/8/docs/api/java/util/Map.html

# Maps – adding and removing items

To add items to the map we can use the **put** method:

```
Map<Integer, Customer> customerDict = new HashMap<Integer, Customer>();

customerDict.put(1,new Customer(1,"Carl") );
customerDict.put(12,new Customer(12,"Eric") );
customerDict.put(252,new Customer(252,"Tore") );
customerDict.put(1021,new Customer(1021,"Max") );
...
```

If the HashMap already contains an item it will be replaced!

To remove items we can use the **remove** method:

```
customerDict.remove(1021);
```

# Maps

To check if a value exist in a map we can use:

```
if(customerDict.containsKey(252))
{
    Customer cust = customerDict.get(252);
    System.out.println("Found " + cust.Name);
}
else
{
    System.out.println("Not found!");
}

//Alternative pattern
Customer cust2 = customerDict.get(252);
if(cust2 != null)
{
    System.out.println("Found " + cust2.Name);
}
else
{
    System.out.println("Not found!");
}
```

Returns true if the item exists

Get always returns null if the item does not exist

# Maps and for-each

We can use **for-each** syntax on a **map**, but not directly.
We have to do it by calling the map's **entrySet** method.

```
for (Map.Entry<Integer, Customer> kvp : customerDict.entrySet()) {

    int key = kvp.getKey();
    Customer value = kvp.getValue();

    System.out.println(String.format("%1$s %2$s", key, value.Name));
}
```

```
1 Carl
12 Eric
252 Tore
1021 Max
...
```

This method returns a collection of type **Set** which contains the keys and values in the set.

## Maps

Important characteristics of a Map:

- A **map** is a collection of **key-value pairs**
- Maps are sometimes called **Dictionaries**
- The key must be unique
- Constant lookup time

The collection to use is entirely dependent on the situation, and there's more than the basic types we've covered here.

## Exercise 21

Let's do exercise 21