Java-MCAL12

-Prof. Anamika Dhawan

Generic

The Java Generics allows us to create a single class, interface, and method that can be used with different types of data (objects).

This helps us to reuse our code.

Note: Generics does not work with primitive types (`int`, `float`, `char`, etc).

Why generics?

1. Code Reusability

With the help of generics in Java, we can write code that will work with different types of data.

```
public <T> void genericsMethod(T data) {...}

Public String checkData(int a){}
```

2. Compile-time Type Checking

The type parameter of generics provides information about the type of data used in the generics code.

```
GenericsClass<Integer> list = new GenericsClass<>();
```

3. Used with Collections

The collections framework uses the concept of generics in Java

Generic class

A class that can be used with any type of data. Such a class is known as Generics Class.

```
class GenericsClass<T> {...}
```

Here, T used inside the angle bracket <> indicates the type parameter.

Class Object Creation:

```
GenericsClass<Integer> intObj = new GenericsClass<>(5);

GenericsClass<String> stringObj = new GenericsClass<>("Java Programming");
```

Generic Method

A method that can be used with any type of data.

```
public <T> void genericMethod(T data) {...}
```

Here, T used inside the angle bracket <> indicates the type parameter

Method Call:

```
<String>genericMethod("Java Programming");

<Integer>genericMethod(25);
```

Wildcards

In generic code, the question mark (?), called the wildcard, represents an unknown type.

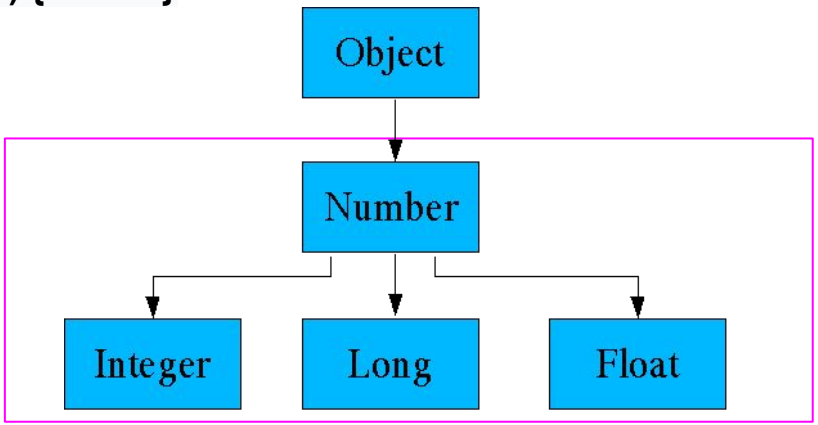
Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.

public static void process(List<? extends Number> list) { /* ... */ }

List list=new ArrayList();

List<Integer>

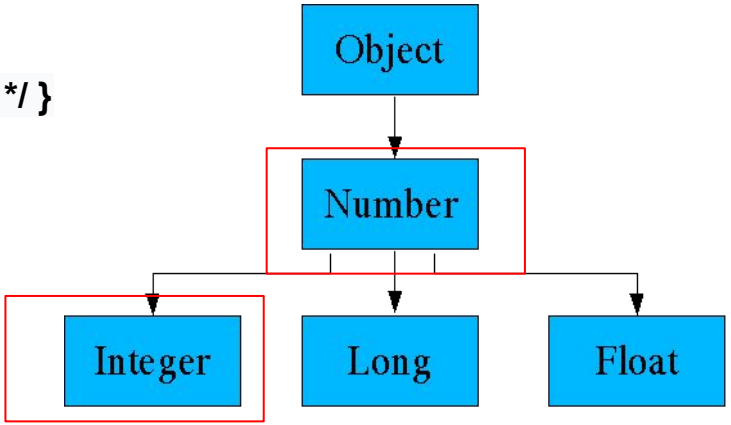


Lower Bounded Wildcards

A lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>.

public static void process(List<? Super Integer> list) { /* ... */ }



Unbounded Wildcards

Note:It's important to note that List<Object> and List<?> are not the same. You can insert an Object, or any subtype of Object, into a List<Object>. But you can only insert null into a List<?>.

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type.

There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the Object class.
- When the code is using methods in the generic class that don't depend on the type parameter. For example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Collection Framework

A collections framework is a unified architecture for representing and manipulating collections.

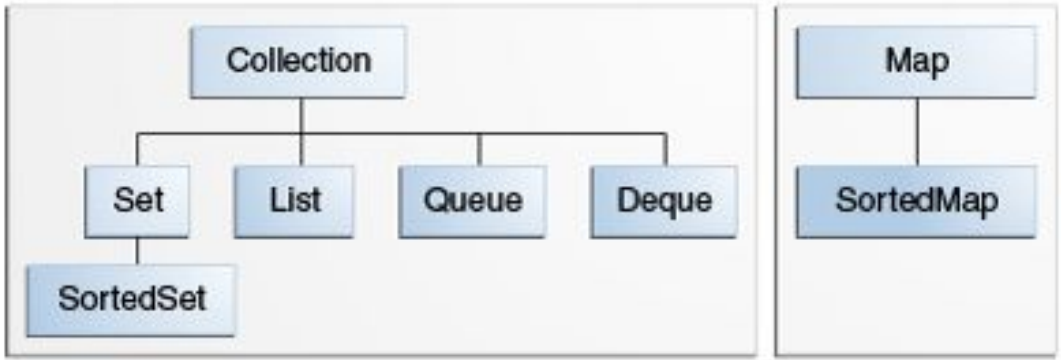
All collections frameworks contain the following:

- Interfaces
- Implementation
- Algorithms

Interfaces

Note that all the core collection interfaces are generic.

The core collection interfaces encapsulate different types of collections.



https://docs.oracle.com/javase/8/docs/api/

Operations

- Basic Operations
- Bulk Operations
- Iteration

Operations

The **Collection** interface includes a variety of methods:

- Adding objects to the collection: **add(E), addAll(Collection)**
- Testing size and membership: **size(), isEmpty(), contains(E), containsAll(Collection)**
- Iterating over members: **iterator()**
- Removing members: **remove(E), removeAll(Collection), clear(), retainAll(Collection)(**
- Generating array representations: **toArray(), toArray(T[])**

Operations

java.util.Iterator is an interface specifying the capabilities of an iterator.

Invoking the iterator() method on a collection returns an iterator object that implements Iterator and knows how to step through the objects in the underlying collection.

The Iterator interface specifies the following methods:

hasNext() - Returns true if there are more elements in the collection; false otherwise

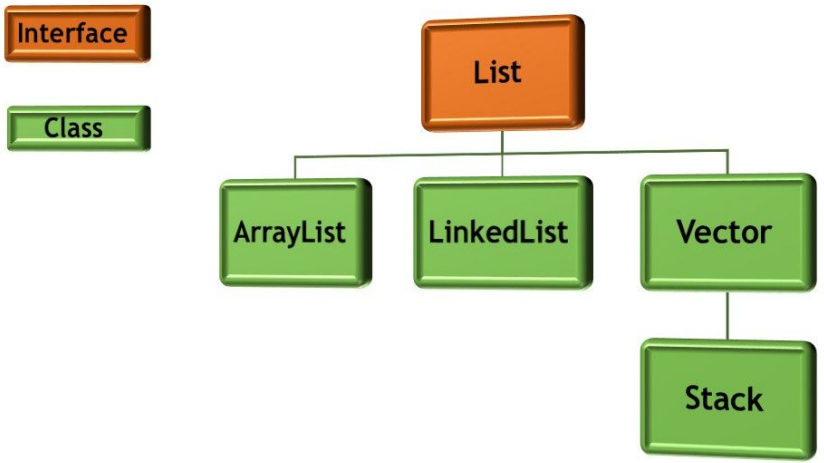
next() - Returns the next element

remove() - Removes from the collection the last element returned by the iterator

List

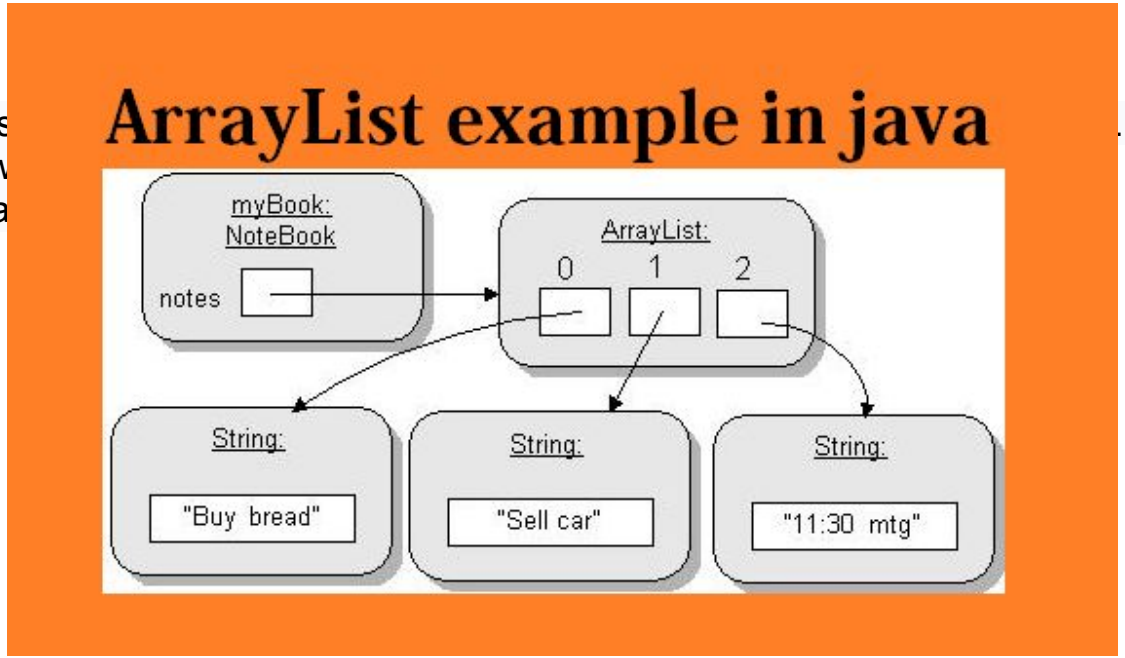
The List interface provides a way to store the ordered collection.
It is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored.

- **List heirarchy**
- **List operations**



ArrayList

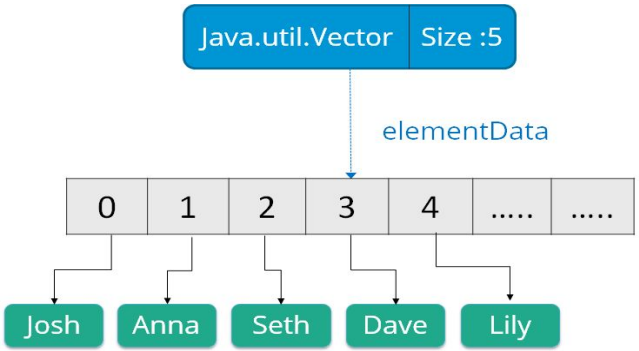
ArrayList class which is
Though, it may be slow
manipulation in the arra

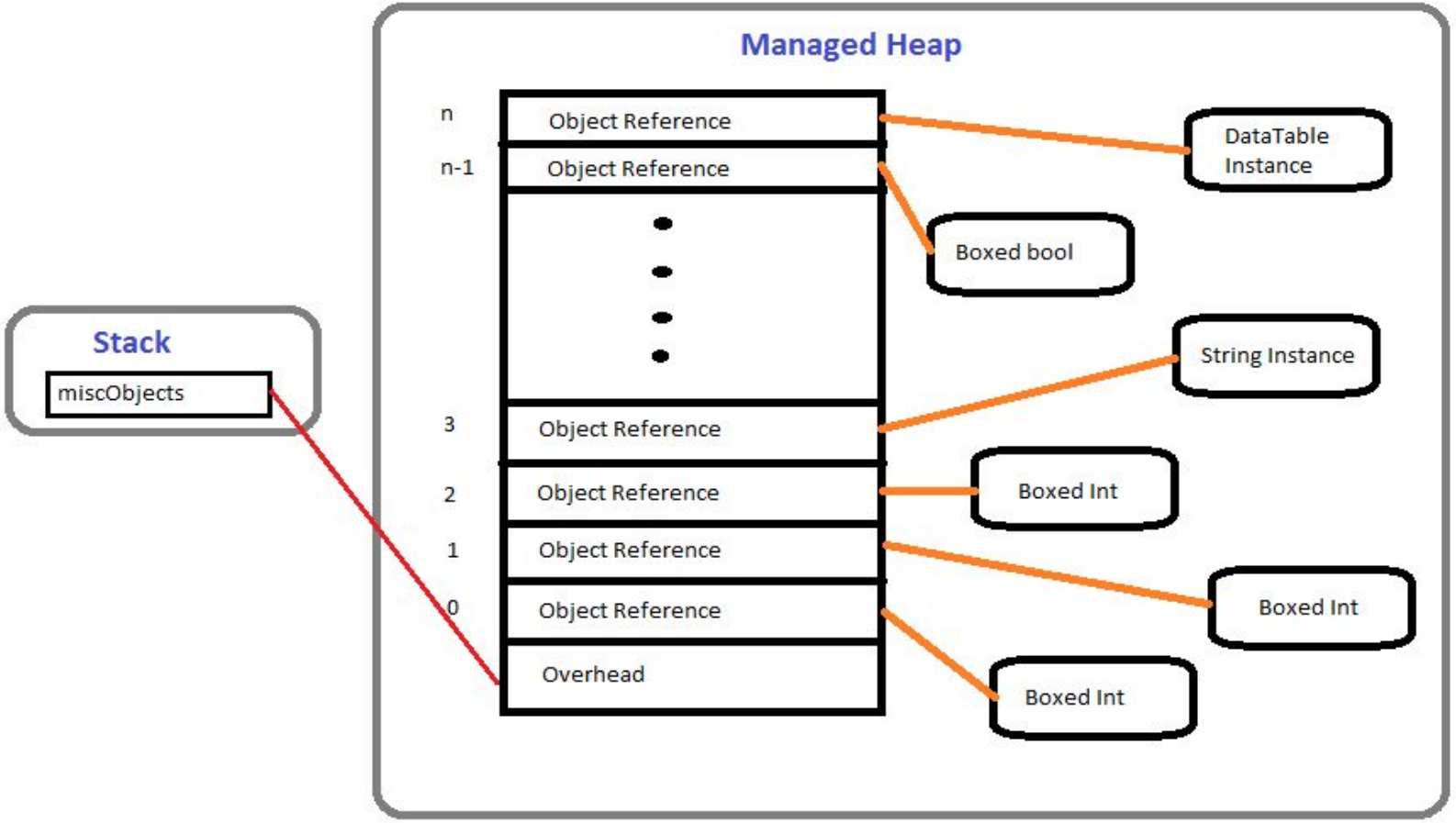


Vector

Vector is a class which is implemented in the collection framework implements a growable array of objects.

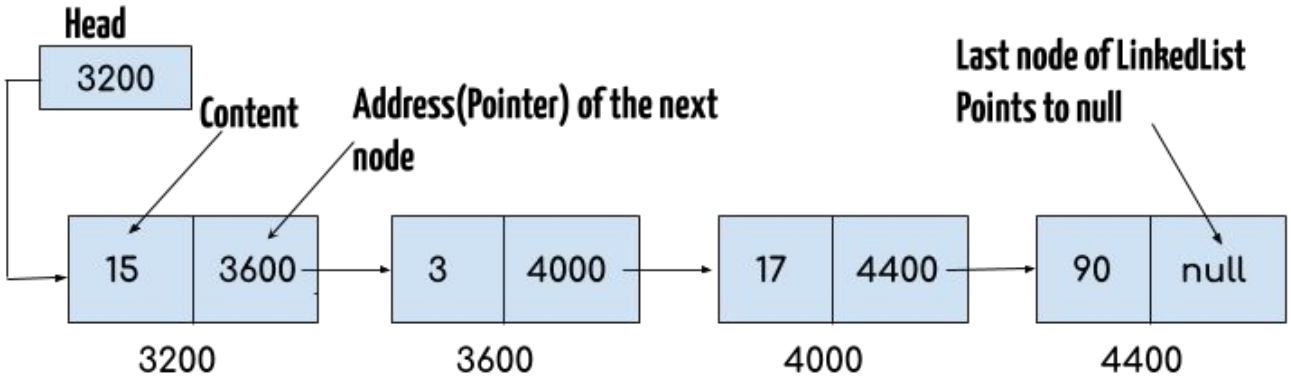
Vector implements a dynamic array that means it can grow or shrink as required.




Managed Heap
n
Object Reference
n-1
Object Reference
DataTable Instance
Boxed bool
String Instance
3
Object Reference
2
Object Reference
Boxed Int
1
Object Reference
Boxed Int
0
Object Reference
Overhead
Boxed Int
Stack
miscObjects

Linked List

LinkedList is a class which is implemented in the collection framework which inherently implements the linked list data structure.

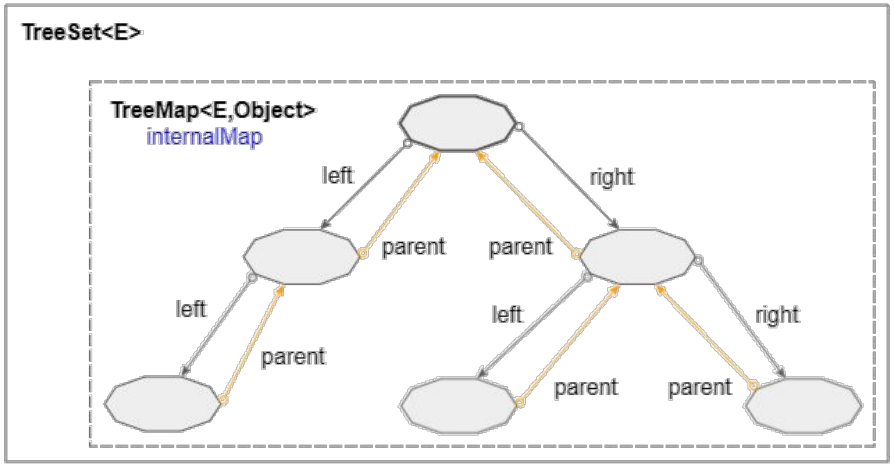


Set

The set interface present in the java.util package and extends the Collection interface is an unordered collection of objects in which duplicate values cannot be stored.
It is an interface which implements the mathematical set.
Set operations.

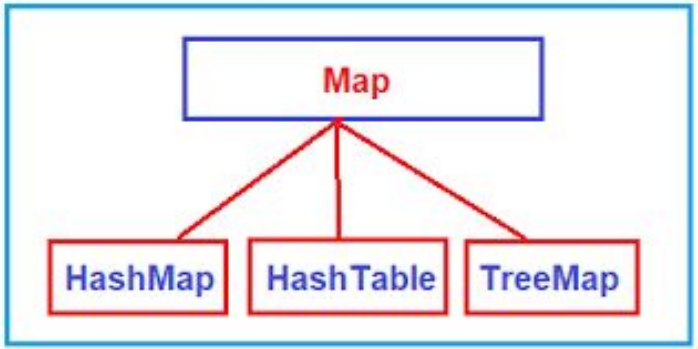
Treeset

TreeSet class which is implemented in the collections framework an implementation of the SortedSet Interface and SortedSet extends Set Interface. It behaves like simple set with the exception that it stores elements in sorted format.



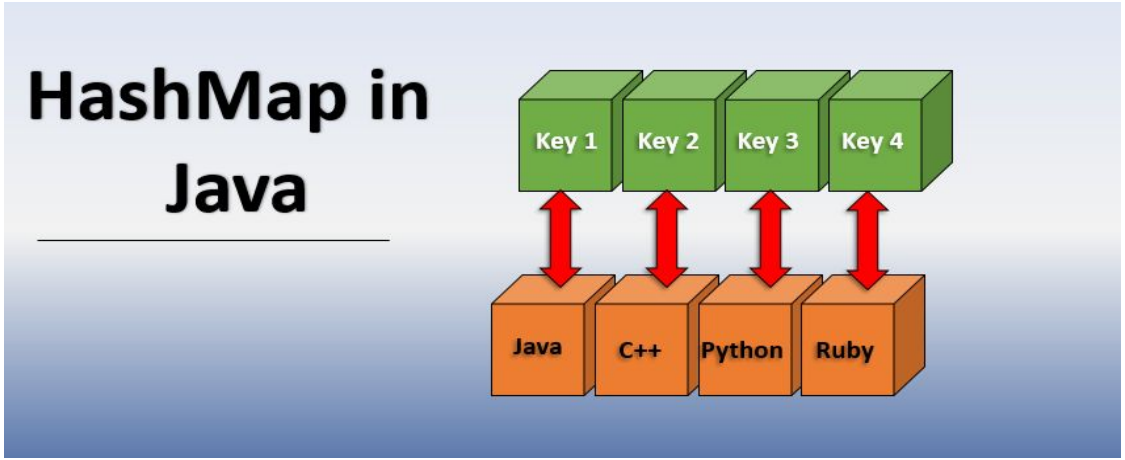
Map

An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.



Hashmap

Hash table based implementation of the Map interface.
This implementation provides all of the optional map operations, and permits null values and the null key.



Lambda Expression

A lambda expression is a short block of code which takes in parameters and returns a value.

Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

The simplest lambda expression contains a single parameter and an expression:

parameter -> expression

To use more than one parameter, wrap them in parentheses:

(parameter1, parameter2) -> expression

Why use lambda?

- To provide the implementation of Functional interface.
- Less coding.

Functional Interface

An interface which has only one abstract method is called functional interface.

Java provides an annotation @FunctionalInterface, which is used to declare an interface as functional interface.

Lambda expression syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

1) **Argument-list**: It can be empty or non-empty as well.

2) **Arrow-token**: It is used to link arguments-list and body of expression.

3) **Body**: It contains expressions and statements for lambda expression.