Programming in Java
Part IV

Carlos Kavka

Head of Research and Development

Agenda

Exceptions

Concurrency

Modularity

Collections

Generics

Exceptions

Exceptions

```
public class TestExceptions1 {
 public static void main(String[] args) {

  String s = "Hello";
  System.out.print(s.charAt(10));
 }
}
```

The usual behavior on runtime errors is to abort the execution

For example, here there is an error in the charAt() call

```
$ java TestExceptions1
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
String index out of range: 10
at java.lang.String.charAt(String.java:499)
at TestExceptions1.main(TestExceptions1.java:11)
```

Exceptions

```
public class TestExceptions2 {
 public static void main(String[] args) {

  String s = "Hello";
  try {
   System.out.print(s.charAt(10));
  } catch (Exception e) {
   System.out.println("No such position");
 }
}
```

The exception can be trapped by using a try-catch block

```
$ java TestExceptions2
No such position
```

Handling exceptions

It is possible to specify interest on a particular exception

```
public class TestExceptions4 {
 public static void main(String[] args) {

  String s = "Hello";
  try {
   System.out.print(s.charAt(10));
  } catch (StringIndexOutOfBoundsException e) {
   System.out.println("No such position");
   System.out.println(e.toString());
 }
}
```

And also send messages to an exception object

```
$ java TestExceptions4
No such position
java.lang.StringIndexOutOfBoundsException:
String index out of range: 10
```

Handling multiple exceptions

```
public static void printInfo(String sentence) {
   try {
      // get first and last char before the dot
      char first = sentence.charAt(0);
      char last = sentence.charAt(sentence.indexOf(".") - 1);
      String out = String.format("First: %c Last: %c",first, last);
      System.out.println(out);

   } catch (StringIndexOutOfBoundsException e1) {
      System.out.println("Wrong sentence, no dot?");

   } catch (NullPointerException e2) {
      System.out.println("Non valid string");

   } finally {
      System.out.println("done!");
   }
}
```

It is possible to add multiple catch blocks and optionally a single finally clause

Handling multiple exceptions

```
public static void printInfo(String sentence) throws {
   try {
    // get first and last char before the dot
    char first = sentence.charAt(0);
    char last = sentence.charAt(sentence.indexOf(".") - 1);
    String out = String.format("First: %c Last: %c",first, last);
    System.out.println(out);

   } catch (StringIndexOutOfBoundsException| NullPointerException e) {
    System.out.println("Non valid string");

   } finally {
      System.out.println("done!");
   }
}
```

Or join catch blocks if the action is the same

Throwing Exceptions

```
import java.io.*;

class WriteFile {
  public static void main(String[] args) {
    FileWriter f;
    BufferedWriter bf;
    try {
      f = new FileWriter("file1.text");
      bf = new BufferedWriter(f);
      String s = "Hello World!";
      bf.write(s,0,s.length());
      bf.newLine();
      bf.write("Java is nice!!!",8,5);
      bf.newLine();
      bf.close();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
  }
}
```

Exception can be handled in the method where it arises …

Throwing Exceptions

```
public class WriteFile {
  public static void main(String[] args) {
    WriteFile wf = new WriteFile();
    try {
      wf.write();
    } catch (IOException e) {
      System.out.println("Error with files:"+e.toString());
    }
  }
  public void write() throws IOException {
    FileWriter f;
    BufferedWriter bf;
    f = new FileWriter("file1.text");
    bf = new BufferedWriter(f);
    String s = "Hello World!";
    bf.write(s,0,s.length());   bf.newLine();
    bf.write("Java is nice!!!",8,5);  bf.newLine();
    bf.close();
  }
}
```

… or can be thrown to the upper level

Creating new Exceptions

```
public class ExceptionTest {
   public static void main(String[] args) {
      ExceptionTest t = new ExceptionTest();
      try {
         t.test(3);
      } catch (MyException e) {
         e.printStackTrace();
      }
   }

   public int test(int x) throws MyException {
      if (x < 0)
         throw new MyException();
      return x - 1;
   }
}
```

New exceptions can be created and integrated into the hierarchy of exceptions

```
public class MyException extends Exception {

}
```

Concurrency

Threads

- ✓ It is possible to run concurrently different tasks called threads.
- ✓ The threads can communicate between themselves
- ✓ Their access to shared data can be synchronized
- ✓ Two implementation possibilities: extend thread or implement runnable interface

Sub-classing the Thread class

```
class CharThread extends Thread {
 char c;
 CharThread(char aChar) {
  c = aChar;
 }
 public void run() {
  while (true) {
   System.out.println(c);
   try {
    sleep(100);
   } catch (InterruptedException e) {
    System.exit(0);
   }
  }
 }
}
```

```
class TestThreads {
 public static void main(String[] args) {
  CharThread t1 = new CharThread('a');
  CharThread t2 = new CharThread('b');

  t1.start();
  t2.start();
 }
}
```

```
$ java TestThreads
a
b
a
b
...
```

Runnable interface

```
class CharThread implements Runnable {
 char c;
 CharThread(char aChar) {
  c = aChar;
 }
 public void run() {
  while (true) {
   System.out.println(c);
   try {
    Thread.sleep(100);
   } catch (InterruptedException e) {
    System.out.println("Interrupted");
   }
  }
 }
}
```

Now the class can extend other classes

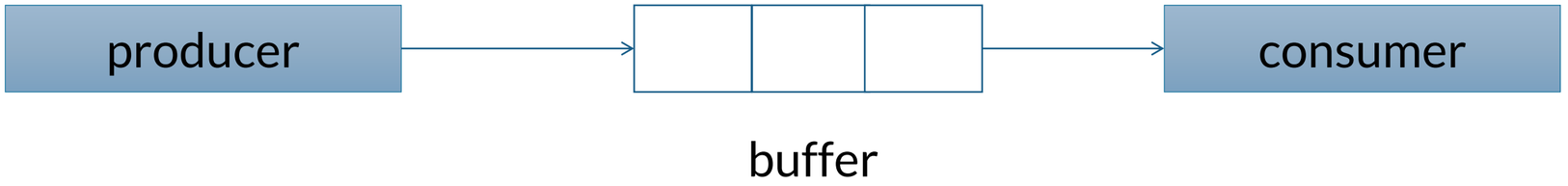
Note that sleep is not inherited any more!

An example

```
class ProducerConsumer {
 public static void main(String[] args) {
  Buffer buffer = new Buffer(10);
  Producer prod = new Producer(buffer);
  Consumer cons = new Consumer(buffer);

 prod.start();
 cons.start();
 }
}
```

The producer and the consumer are implemented with threads

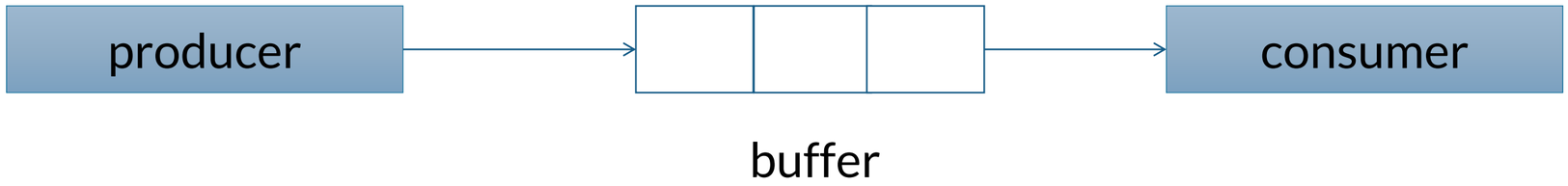
The buffer is shared between the two threads



An example: the producer and consumer

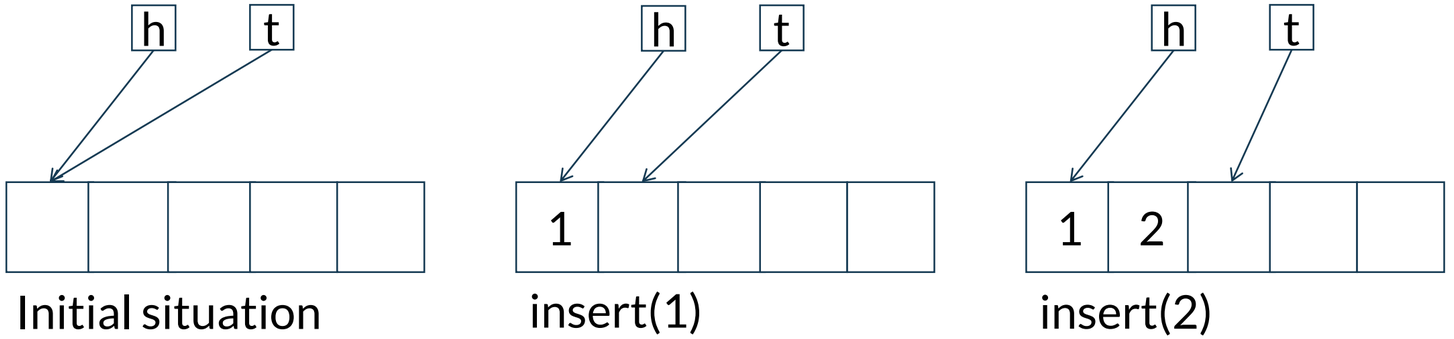
```
class Producer extends Thread {
 Buffer buffer;

 public Producer(Buffer b) {
  buffer = b;
 }

 public void run() {
  double value = 0.0;
  while (true) {
   buffer.insert(value);
   value += 0.1;
  }
 }
}
```

```
class Consumer extends Thread {
 Buffer buffer;

 public Consumer(Buffer b) {
  buffer = b;
 }

 public void run() {
  while(true) {
   double element = buffer.delete()
   System.out.println(element);
  }
 }
}
```

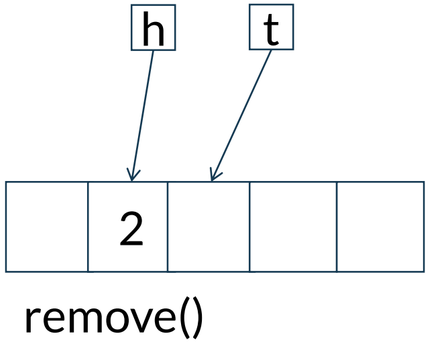


An example: the circular buffer

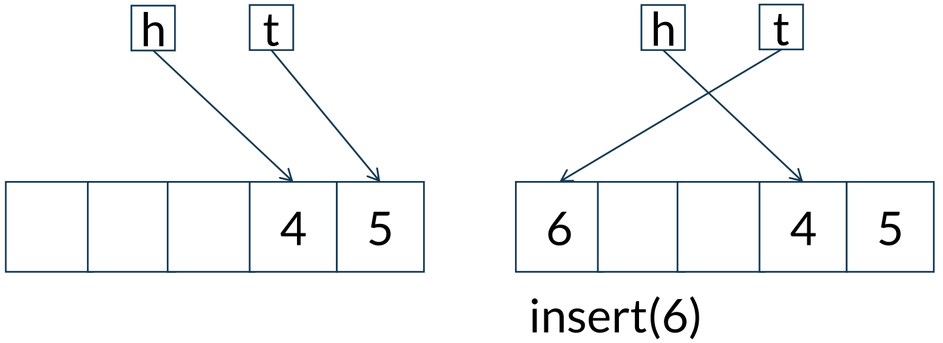
Insertion of elements in the buffer:



Remove one element:



Going beyond the limit of the buffer:



An example: the buffer

```
class Buffer {
 private double []buffer;
 private int head = 0,tail = 0,size = 0,numElements = 0;

 public Buffer(int s) {
  buffer = new double[s];
  size = s;
 }
 public void insert(double element) {
  buffer[tail] = element; tail = (tail + 1) % size;
  numElements++;
 }
 public double delete() {
  double value = buffer[head]; head = (head + 1) % size;
  numElements--;
  return value;
 }
}
```

An example: problems

However, the implementation does not work!.

- The methods insert() and delete() operate concurrently over the same structure.
- The method insert() does not check if there is at least one slot free in the buffer
- the method delete() does not check if there is at least one piece of data available in the buffer.

There is a need for synchronization

Synchronization

- ✓ Synchronized access to a critical resource can be achieved with synchronized methods
- ✓ Each instance has a lock, used to synchronize the access.
- ✓ Synchronized methods are not allowed to be executed concurrently on the same instance.

An example: synchronized methods

```
public synchronized void insert(double element) {

while (numElements == size) {
   try {
    wait();
   } catch(InterruptedException e) {
    System.out.println("Interrupted");
   }
 }

 buffer[tail] = element;
 tail = (tail + 1) % size;
 numElements++;
 notify();
}
```

The methods goes to sleep (and release the lock) if buffer is full

At the end, it awakes producer(s) which can be sleeping waiting for the lock

Synchronized access to the critical resource is achieved by defining the methods as synchronized

Please not that a **while** is used and not an **if.** Why?

An example: synchronized methods

```
public synchronized double delete() {

 while (numElements == 0) {
  try {
   wait();
  } catch(InterruptedException e) {
   System.out.println("Interrupted");
  }
 }

 double value = buffer[head];
 head = (head + 1) % size;
 numElements--;
 notify();
 return value;
}
```

The methods goes to sleep (and release the lock) if buffer is empty

At the end, it awakes consumer(s) which can be sleeping waiting for the lock

Synchronized access to the critical resource is achieved by defining the methods as synchronized

Concurrency

A more powerful and modern approach to concurrency based on streams will be discussed later on.

Modularity

Packages

A package is a structure in which classes can be organized.

It can contain any number of classes, usually related by purpose or by inheritance.

The standard classes in the system are organized in packages

```
import java.util.*; // or import java.util.Date

public class TestDate {
  public static void main(String[] args) {
    System.out.println(new Date());
  }
}
```

Packages

Package name is defined by using the keyword package as the first instruction

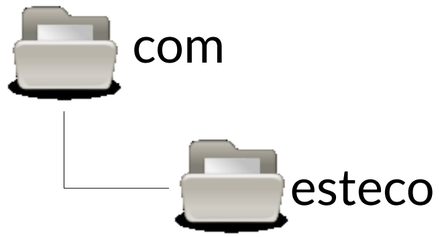
```
package myBook;

public class Book {
  String title;
  String author;
  int numberOfPages;
}
```

```
package myBook;

public class ExampleBooks {
  public static void main(String[] args) {

    Book b = new Book();
    b.title = "Java 8 Lambdas";
    b.author = "Richard Warburton";
    b.numberOfPages = 168;
    System.out.println(b.title + " : " + b.author + " : " + b.numberOfPages);
  }
}
```

Packages

```
package com.esteco;

public class Book {
  String title;
  String author;
  int numberOfPages;
}
```

```
package com.esteco;

public class ExampleBooks {
  public static void main(String[] args) {

    Book b = new Book();
}
```



There is a correspondence between the package name and the directory structure where the classes are located

Modules

- ✓ Modules describe code dependencies and relations in an application
- ✓ Convenient for large applications
- ✓ The Java run-time library is structured in modules as seen in the Java API web definition. For example: java.base, java.desktop, java.xml, etc.

A simple example with modules

In each module, the file module-info.java describes the relationships and dependencies

```
package com.esteco;

public class Book {
  String title;
  String author;
  int numberOfPages;
}
```

```
package it.units;

import com.esteco.Book;

public class Film {
  String title;
  String director;
  Book basedOn;
}
```

```
module modulebooks {
   exports com.esteco;
}
```

module-info.java 

```
module modulefilms {
   requires modulebooks;
}
```

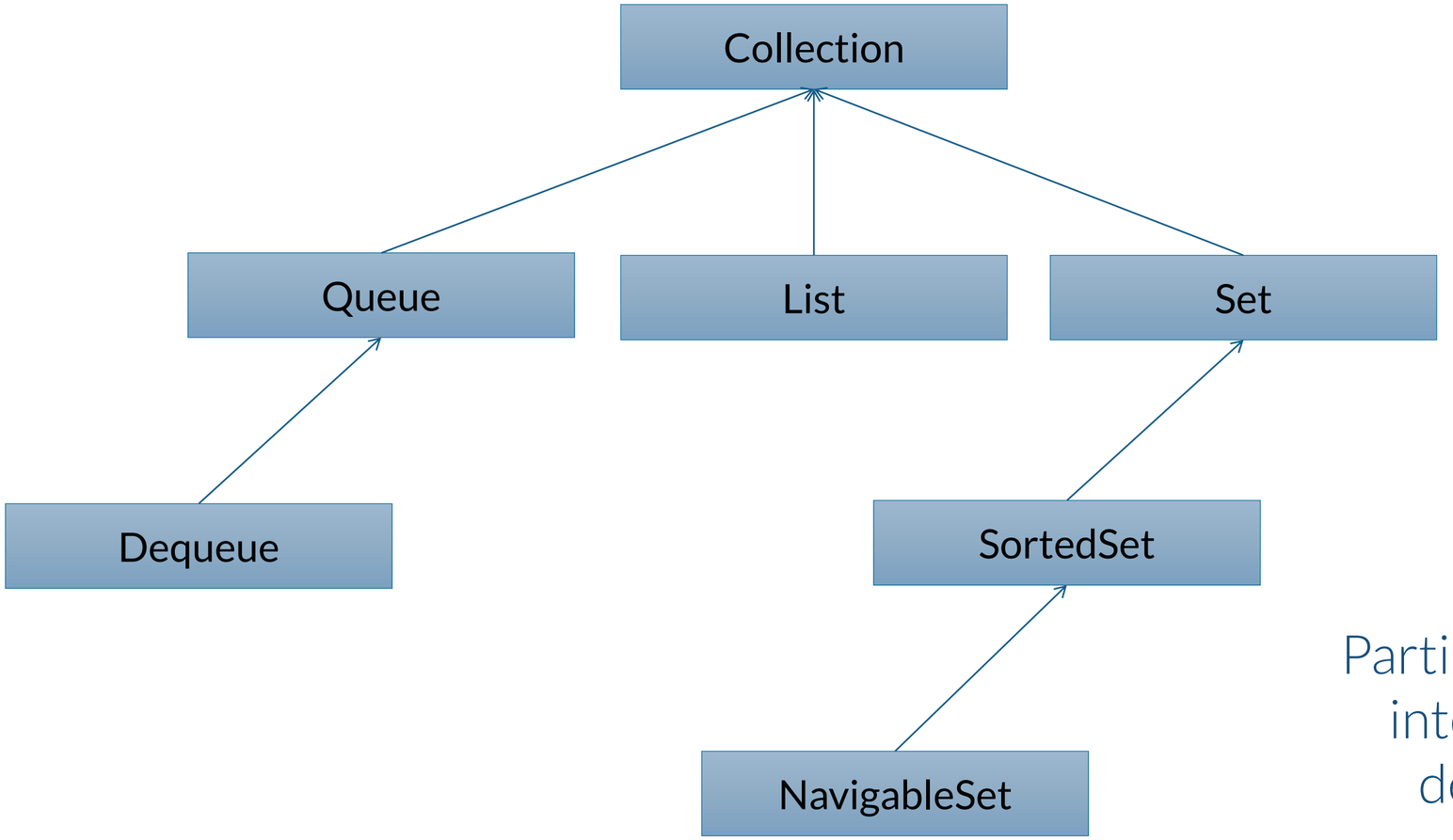
module-info.java 

Collections

Collections Framework

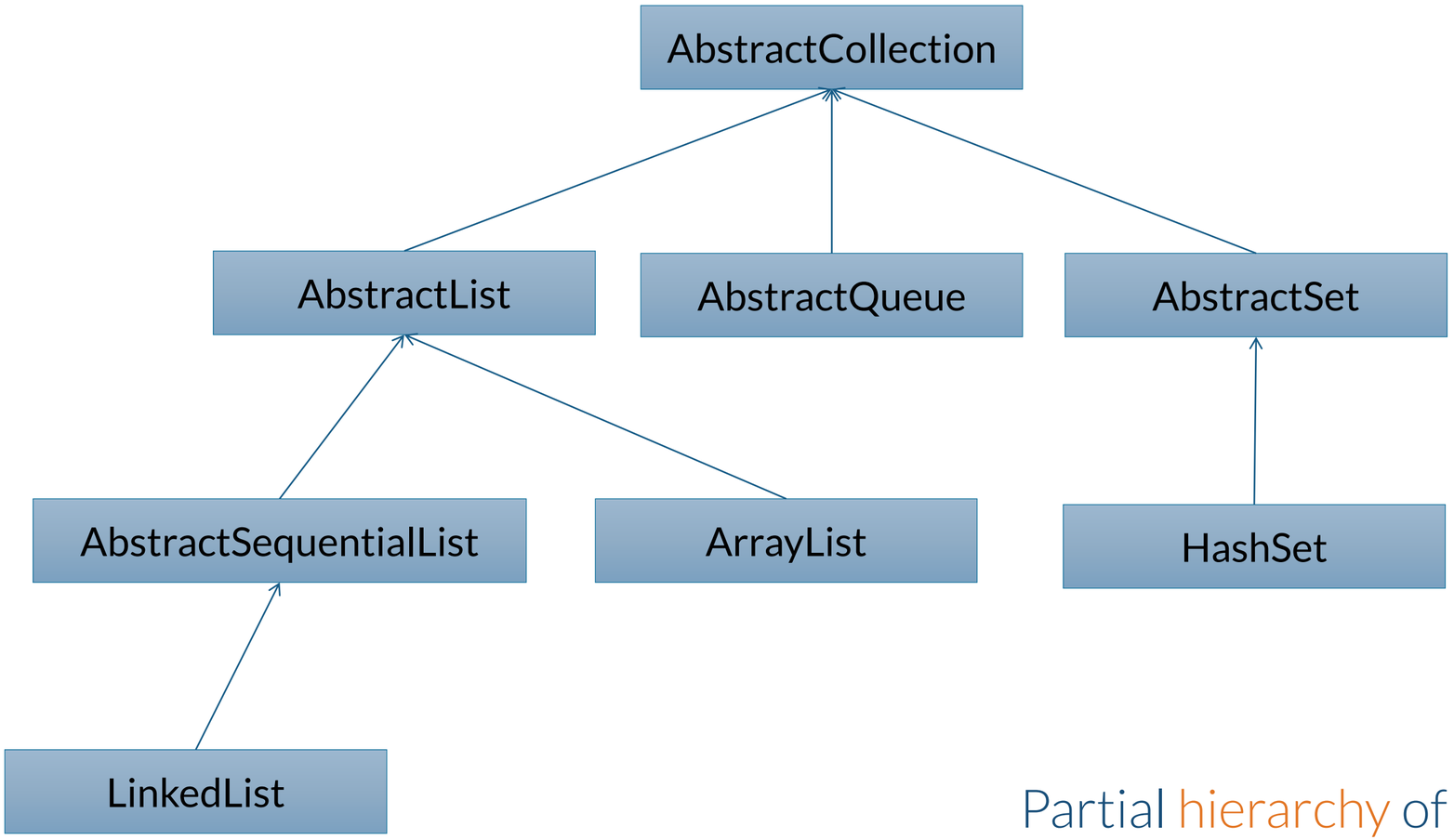
- ✓ The framework provides state-of-the-art technology for managing groups of objects
- ✓ A highly sophisticated hierarchy of interfaces and classes
- ✓ Java programmers must know and use it

Interfaces



Partial hierarchy of interfaces, which define behavior

Classes



Partial hierarchy of classes, which define implementation

An example with ArrayList

Creation and insertion

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

list.add("red");
list.add("blue");
list.add("white");
```

Traversing the structure

```
for(String x : list) {
   System.out.println(x);
}
```

Removing elements

```
list.remove(2);
list.remove("white");
```

An example with LinkedList

Creation and insertion

```
LinkedList<String> list = new LinkedList<>();

list.add("red");
list.addFirst("blue");
list.add(1,"white");
```

Please note the use of the diamond <> notation

```
for(String x : list) {
   System.out.println(x);
}
```

Traversing the structure

Removing elements

```
list.last();
list.remove("white");
```

An example with HashMap

Creation and insertion

```
HashMap<String, Integer> map = new HashMap<>();

map.put("temperature", 22);
map.put("humidity", 65);
```

```
int temp = map.get("temperature");
```

Accessing a value

Getting keys

```
for(String x : map.keySet()) {
    System.out.println(map.get(x));
}
```

Collections

A more powerful and modern approach to deal with data stored in Collections based on the streams will be discussed later on.

Generics

Generics

- ✓ Generics allows to build parameterized types
- ✓ create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- ✓ Improve type safety when compared with the use of Objects

An example

```
public class Stack<E> {
   private LinkedList<E> data;

   Stack() {
      data = new LinkedList<E>();
   }

   public void push(E x) {
      data.addFirst(x);
   }

   public E pop() {
      return data.removeFirst();
   }

 public int size() {
      return data.size();
   }
}
```

A generic stack

```
   public static void main(String[] args) {
      Stack<Integer> stack = new Stack<>();
      stack.push(22);
      stack.push(66);
      System.out.println(stack.pop());
   }
}
```

Be careful with this small example, no checking is done when removing elements!

Bounded classes

The generic class can be restricted

```
public class Stack<BaseType extends Number> {
  …
}
```

This specifies that BaseType can only be replaced by Number,
or subclasses of Number.

Wildcard arguments

Let’s defined a new methods to compare the size of two stacks:

```
public class Stack<BaseType extends Number> {
…
   public boolean equalSize(Stack<BaseType> other) {
      return size() == other.size();
   }
}
```

It seems simple…

```
Stack<Integer> stack1 = new Stack<>();
stack1.push(22);
stack1.push(66);

Stack<Float> stack2 = new Stack<>();
stack2.push(3.1F);

boolean equalSize = stack1.equalSize(stack2);
```

…however, it does not work if types are different!

Wildcard arguments

A new method with wildcards to compare the size of two stacks:

```
public class Stack<BaseType extends Number> {
…
   public boolean equalSize(Stack<?> other) {
      return size() == other.size();
   }
}
```

```
Stack<Integer> stack1 = new Stack<>();
stack1.push(22);
stack1.push(66);

Stack<Float> stack2 = new Stack<>();
stack2.push(3.1);

boolean equalSize = stack1.equalSize(stack2);
```

It works now!!!!

Comparator interface for Collections

Classes that implements the comparable interface can be "compared" by Collection methods

```
public interface Comparable<T extends Object> {

   public int compareTo(T t);
}
```

Note the bounded generic declaration!

Comparator interface for Collections

```
class Book implements Comparable<Book> {
  ...
  public int compareTo(Book aBook) {
     return numberOfPages - aBook.numberOfPages;
   }
```

With the comparator,
Books can be compared
now!

```
Book b1 = new Book ("Java 8 Lambdas",""Richard Warburton",168);
Book b2 = new Book("Java in a nutshell", "David Flanagan",353);

ArrayList<Book> list = new ArrayList<Book>();

list.add(b1);    list.add(b2);

Collections.sort(list);

for (Book x : list) {
  System.out.println(x.title);
}
```

Thank you!

esteco.com