JC2002 Java Programming

Day 11: Strings and collections (AI, CS)

Wednesday, 15 November / Thursday, 16 November

JC2002 Java Programming

Day 11, Session 1: Strings and basic string operations

References and learning objects

- Today's sessions are largely based on ***Java: How to Program***, Chapter 14, and ***Java in a Nutshell***
- After today's session, you should be able to:
  - Use strings, string builders, and string operations in Java programs
  - Use basic regular expression (RegExp) operations
  - Use list and iterators to implement collections in Java programs

Introduction to strings

- In many Java programs, strings and string operations are essential
  - An instance of class `String` represents a string, i.e., a sequence of characters
  - Class `String` provides several methods to create and manipulate strings: we have already used some basic string operations in the earlier sessions
- `String` objects are immutable
  - The contents of `String` objects cannot be changed after creation; this is why many String methods actually create a copy of the original `String` that is manipulated

String constructors (1)

- Class `String` provides constructors for initialising String objects in a variety of different ways:
  - Without an argument, an empty string is created; however, since strings are immutable, empty strings are usually worthless:

```
String s0 = new String();
```

s0 = “”

  - You can use a constant string as an argument:

```
String s1 = new String("hello");
```

s1 = “hello”

  - You can use a `String` object as an argument to create a copy:

```
String s2 = new String(s1);
```

s2 = “hello”

String constructors (2)

- Class `String` provides also constructors accepting character or byte arrays as arguments:

```
char[] charArray = {'b','i','r','t','h',' ','d','a','y’};

String s3 = new String(charArray);

String s4 = new String(charArray, 6, 3);
```

s3 = “birth day”

s4 = “day”

The starting position (offset) where the characters in the array are accessed

The number of characters (count) to access

Initialising strings as literals

- In Java, `String` objects can be also created by assigning a *string literal* without keyword `new`:

```
String s = "hello";
```

- It should be noted that keyword `new` creates *always* a new `String` object, whereas strings created by literal will refer to an existing object, if similar string exists in the pool of string literals already
  - Since `String` objects are immutable, the difference is practically insignificant

UNIVERSITY OF ABERDEEN

Example of string literals (1)

```java
public class StringLiteralExample {
  public static void main(String[] args){
    String s1 = "hello";
    String s2 = new String("hello");
    String s3 = "hello";
    System.out.println("Are s1 and s2 same? " + (s1 == s2));
    System.out.println("Are s1 and s3 same? " + (s1 == s3));
    System.out.println("Are s2 and s3 same? " + (s2 == s3));
  }
}
```

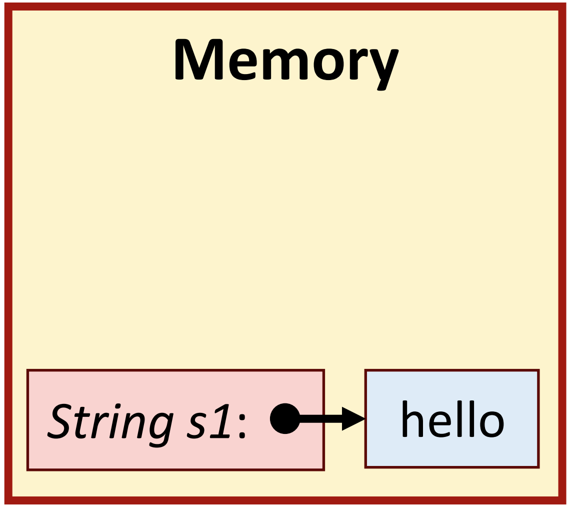
```
$ java StringLiteralExample
```

Memory

Example of string literals (2)

```java
public class StringLiteralExample {
  public static void main(String[] args){
    String s1 = "hello";
    String s2 = new String("hello");
    String s3 = "hello";
    System.out.println("Are s1 and s2 same? " + (s1 == s2));
    System.out.println("Are s1 and s3 same? " + (s1 == s3));
    System.out.println("Are s2 and s3 same? " + (s2 == s3));
  }
}
```

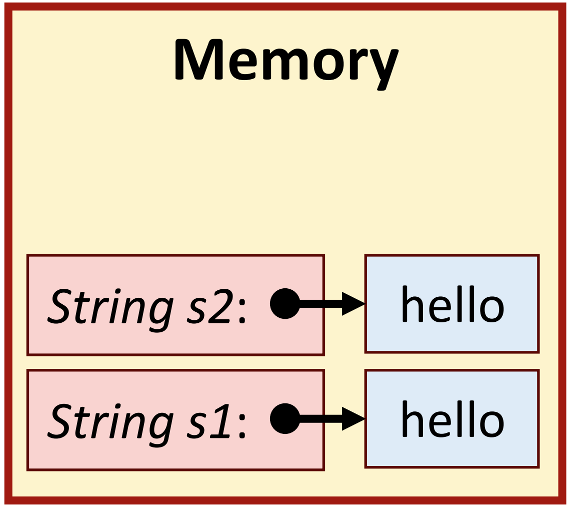
```
$ java StringLiteralExample
```



Example of string literals (3)

```java
public class StringLiteralExample {
  public static void main(String[] args){
    String s1 = "hello";
    String s2 = new String("hello");
    String s3 = "hello";
    System.out.println("Are s1 and s2 same? " + (s1 == s2));
    System.out.println("Are s1 and s3 same? " + (s1 == s3));
    System.out.println("Are s2 and s3 same? " + (s2 == s3));
  }
}
```

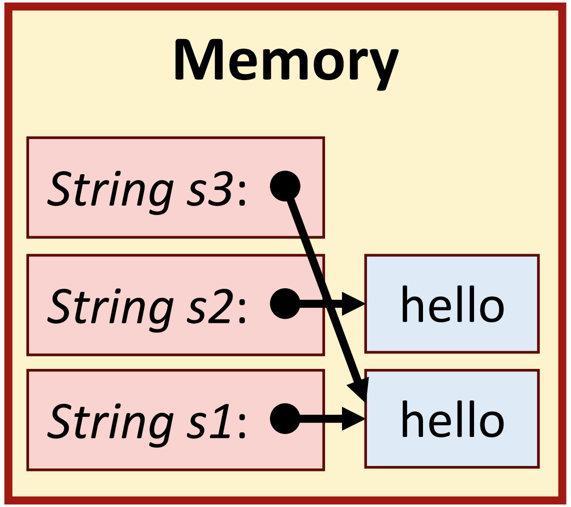
```
$ java StringLiteralExample
```



Example of string literals (3)

```java
public class StringLiteralExample {
  public static void main(String[] args){
    String s1 = "hello";
    String s2 = new String("hello");
    String s3 = "hello";
    System.out.println("Are s1 and s2 same? " + (s1 == s2));
    System.out.println("Are s1 and s3 same? " + (s1 == s3));
    System.out.println("Are s2 and s3 same? " + (s2 == s3));
  }
}
```

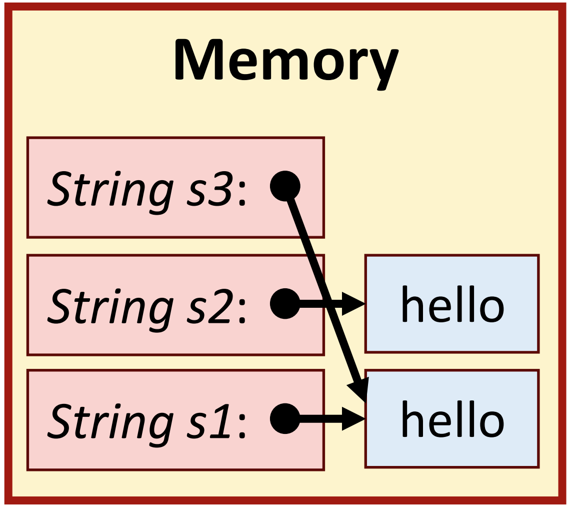
```
$ java StringLiteralExample
```



Example of string literals (4)

```java
public class StringLiteralExample {
  public static void main(String[] args){
    String s1 = "hello";
    String s2 = new String("hello");
    String s3 = "hello";
    System.out.println(“Are s1 and s2 same? " + (s1 == s2));
    System.out.println(“Are s1 and s3 same? " + (s1 == s3));
    System.out.println(“Are s2 and s3 same? " + (s2 == s3));
  }
}
```

```
$ java StringLiteralExample
Are s1 and s2 same? false
Are s1 and s3 same? true
Are s2 and s3 same? false
$
```



Note that comparison (==) applies to the pointers, not the data they point to!

Basic String methods

- The basic `String` methods include the following:
  - `int` **`length()`**: returns the length (number of characters in the string)
  - `char` **`charAt(`**`pos`**`):`** returns the character at the position given in argument pos of type `int` (note that the first character is in position 0)
  - `void` **`getChars(`**`beg,end,dest,destBeg`**`):`** copies characters from the string to the character array
    - `int beg:` the index (position) where copying starts
    - `int end:` the index (position) *next to the last character* to be copied
    - char[] dest: the character array where the characters are copied
    - `int destBeg:` the index (position) in `dest` where copying in starts

Basic String methods example (1)

```java
public class BasicStringExample {
  public static void main(String[] args){
    String str = "hello world!";
    for(int i=str.length()-1; i >= 0; i--) {
        System.out.printf("%c", str.charAt(i));
    }
    System.out.println();
    char[] charArr = new char[5];
    str.getChars(6,11,charArr,0);
    System.out.println(charArr);
  }
}
```

Basic String methods example (2)

```java
public class BasicStringExample {
  public static void main(String[] args){
    String str = "hello world!";
    for(int i=str.length()-1; i >= 0; i--) {
        System.out.printf("%c", str.charAt(i));
    }
    System.out.println();
    char[] charArr = new char[5];
    str.getChars(6,11,charArr,0);
    System.out.println(charArr);
  }
}
```

```
$ java BasicStringExample
!dlrow olleh
```

Loops through the characters backwards and prints them one by one. Note that the last character is at index position `length()-1`.

UNIVERSITY OF ABERDEEN

Basic String methods example (3)

```java
public class BasicStringExample {
  public static void main(String[] args){
    String str = "hello world!";
    for(int i=str.length()-1; i >= 0; i--) {
        System.out.printf("%c", str.charAt(i));
    }
    System.out.println();
    char[] charArr = new char[5];
    str.getChars(6,11,charArr,0);
    System.out.println(charArr);
  }
}
```

```
$ java BasicStringExample
!dlrow olleh
World
$
```

Copies characters from position index 6 to position index 10 of `str` to the character array `charArr`.

UNIVERSITY OF ABERDEEN

Comparing strings

- Note that the Java comparison operator == compares the references, not the contents of the strings
- For comparing the contents of two strings, methods **`equals()`** and **`compareTo()`** can be used
    - Method `equals()` returns true if the argument string contains the same sequence of characters as this object
    - Method `equalsIgnoreCase()` is like `equals()`, but ignores case
    - Method `compareTo()` returns a negative integer if this string lexicographically (alphabetically) precedes the argument string, zero if the strings are equal, and positive integer if this string lexicographically follows the argument string

String comparison example

```java
public class BasicStringComparisonExample {
  public static void main(String[] args){
    String s1 = "albert";
    String s2 = "Albert";
    String s3 = "Bertha";
    System.out.printf("%s equals %s: %b%n", s1, s2, s1.equals(s2));
    System.out.printf("%s equalsIgnoreCase %s: %b%n", s1, s2,
                                s1.equalsIgnoreCase(s2));
    System.out.printf("%s compareTo %s: %d%n", s2, s3, s2.compareTo(s3));
    System.out.printf("%s compareTo %s: %d%n", s3, s2, s3.compareTo(s2));
  }
}
```

```
$ java BasicStringComparisonExample
albert equals Albert: false
albert equalsIgnoreCase Albert: true
Albert compareTo Bertha: -1
Bertha compareTo Albert: 1
$
```

Comparing string regions (substrings)

- For comparing *regions* of strings rather than full strings, method **`regionMatches()`** can be used
  - Returns true if the substrings in specified regions are equal
- Two versions with either four or five arguments:
  - Method `regionMatches(off1,str2,off2,len)` returns true if the substrings of length `len` starting at position `off1` in this string and at position `off2` in argument string `str2` are equal
  - In method `regionMatches(ignoreCase,off1,str2,off2,len)` there is an additional first argument of type `boolean` to determine whether the case should be ignored

Extracting substrings from a string

- For extracting substrings, method **`substring()`** can be used
  - Returns a new `String` object created by copying part of an existing `String` object
- Two versions with either one or two arguments:
  - Method `substring(start)` returns the substring starting from position index `start` and ending in the last character of the string
  - Method `substring(start,end)` returns the substring starting from position index `start` up to, *but not including*, the position index end

Using regionMatches() and substring()

```java
public class RegionMatchesExample {
  public static void main(String[] args){
    String s1 = "Hello World!";
    String s2 = "good morning world!";
    System.out.println("Regions matching: " +
                       s1.regionMatches(6,s2,13,6));
    System.out.println("Regions matching with case ignored: " +
                       s1.regionMatches(true,6,s2,13,6));
    System.out.println("Substring of s1 from 6 to end: " + s1.substring(6));
    System.out.println("Substring of s2 from 13 to 17: " + s2.substring(13,18));
  }
}
```

```
$ java RegionMatchesExample
Regions matching: false
Regions matching with case ignored: true
Substring of s1 from 6 to end: World!
Substring of s2 from 13 to 17: world
$
```

Questions, comments?

JC2002 Java Programming

Day 11, Session 2: Advanced string and character operations

Tokenising strings

- It is often useful to break up long strings into smaller pieces, or *tokens*, for example to extract individual words from a sentence
  - This process is called *tokenisation*
- Method **`split()`** of class `String` breaks a string into its components (tokens)
  - Tokens are separated from one another by delimiters, typically white-space characters such as *space*, *tab*, *newline*, and *carriage return*
  - Other characters can also be used as delimiters to separate tokens
  - The arguments for `split()` include delimiting *regular expression* and the optional maximum limit for the number of tokens

Tokenising example 1

```java
import java.util.Scanner;
public class TokenizingExample1 {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter a sentence and press Enter");
    String sentence = scanner.nextLine();
    String[] tokens = sentence.split(" ");
    System.out.printf("Number of tokens: %d%n", tokens.length);
    System.out.println("The tokens are:");
    for (String token : tokens) {
      System.out.println(token);
    }
  }
}
```

Delimiter is space “ ”

```
$ java TokenizingExample1
Enter a sentence and press Enter:
I like Java
Number of elements: 3
The tokens are:
I
like
Java
$
```

Tokenising example 2

```java
import java.util.Scanner;
public class TokenizingExample2 {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter your email address:");
    String sentence = scanner.nextLine();
    String[] tokens = sentence.split("@",2);
    System.out.printf("Your user name is: %s%n", tokens[0]);
    System.out.printf("Your URL is: %s%n", tokens[1]);
  }
}
```

Delimiter is “@”, maximum of 2 tokens are extracted

```
$ java TokenizingExample2
Enter your email address:
jamesbond007@abdn.ac.uk
Your user name is: jamesbond007
Your URL is: abdn.ac.uk
$
```

Concatenating strings

- Method **`concat()`** of `String` can be used to concatenate two `String` objects into a new `String` object containing characters from both strings
  - Syntax: `s1.concat(s2)` will concatenate `String` objects `s1` and `s2` so that `s2` appears after `s1`
- In Java, addition operator + used on `String` objects is defined to perform concatenation
  - Assuming that `s1`, `s2`, and `s3 are string objects`:

    `s1+s2` equals to `s1.concat(s2)`

    `s1+s2+s3` equals to `s1.concat(s2).concat(s3)`

Concatenation example

```java
public class ConcatenationExample {
  public static void main(String[] args){
    String s1 = "Good ";
    String s2 = "morning ";
    String s3 = "world!";
    System.out.println(s1+s2+s3);
    System.out.println(s1.concat(s2).concat(s3));
    int age = 18;
    System.out.println("Michael is " + age + " years old");
  }
}
```

```
$ java ConcatenationExample
Good morning world!
Good morning world!
Michael is 18 years old
$
```

Note that if the data type is not a `String` object, Java converts it automatically to a string representation when operator + is used, but it does not work with `concat()`!

Using StringBuilder for modifiable strings

- In programs that frequently perform string concatenations or other string modifications, it is often more efficient to use class **`StringBuilder`** instead of class `String`
  - `StringBuilder` is a “modifiable” version of `String`: it provides methods such as **`append()`**, **`insert()`**, and **`delete()`** to modify the contents of the string it contains
  - When a `StringBuilder` object is modified, it does not return a new `StringBuffer` object but changes the contents of the original one

StringBuilder constructors

- `StringBuilder` class provides several different constructors:
  - **`StringBuilder()`**: constructs a string builder with no characters and the initial capacity of 16 characters
  - **`StringBuilder(`**`CharSequence seq`**`)`**: constructs a string builder that contains the same characters as the `CharSequence` object `seq`
  - **`StringBuilder(`**`int capacity`**`)`**: constructs a string builder with no characters and the initial capacity specified by the `capacity` argument
  - **`StringBuilder(`**`String str`**`)`**: constructs a string builder initialised to the contents of the `String` argument `str`

StringBuilder methods

- Some of the most essential `StringBuilder` classes are the following:
  - **`length()`**, **`setLength(`**`int length`**`)`**: returns the length (character count), and sets the length, respectively
  - **`capacity()`** and **`ensureCapacity()`**: returns the current capacity, and increases capacity if below the specified minimum capacity, respectively
  - **`charAt()`**, **`setCharAt()`**, **`getChars()`**: getting and setting characters at specified positions
  - **`append()`**, **`insert()`**, **`delete()`**, **`deleteCharAt()`**: modifying the string builder by appending, inserting, and deleting content; there are several overloaded versions of these methods to support different data types

String builder example (1)

```java
public class StringBuilderExample1 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s | length = %d | capacity = %d%n",
        sb.toString(), sb.length(), sb.capacity());
    sb.ensureCapacity(75);
    System.out.printf("New capacity = %d%n", sb.capacity());
    sb.setLength(10);
    System.out.printf("New buffer = %s | new length = %d%n",
        sb.toString(), sb.length());
  }
}
```

String builder example (2)

```java
public class StringBuilderExample1 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s | length = %d | capacity = %d%n",
        sb.toString(), sb.length(), sb.capacity());
    sb.ensureCapacity(75);
    System.out.printf("New capacity = %d%n", sb.capacity());
    sb.setLength(10);
    System.out.printf("New buffer = %s | new length = %d%n",
        sb.toString(), sb.length());
  }
}
```

```
$ java StringBuilderExample1
Buffer = Good morning world! | length = 19 | capacity = 35
```

String builder example (3)

```java
public class StringBuilderExample1 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s | length = %d | capacity = %d%n",
        sb.toString(), sb.length(), sb.capacity());
    sb.ensureCapacity(75);
    System.out.printf("New capacity = %d%n", sb.capacity());
    sb.setLength(10);
    System.out.printf("New buffer = %s | new length = %d%n",
        sb.toString(), sb.length());
  }
}
```

```
$ java StringBuilderExample1
Buffer = Good morning world! | length = 19 | capacity = 35
New capacity = 75
```

String builder example (4)

```
public class StringBuilderExample1 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s | length = %d | capacity = %d%n",
        sb.toString(), sb.length(), sb.capacity());
    sb.ensureCapacity(75);
    System.out.printf("New capacity = %d%n", sb.capacity());
    sb.setLength(10);
    System.out.printf("New buffer = %s | new length = %d%n",
        sb.toString(), sb.length());
  }
}
```

```
$ java StringBuilderExample1
Buffer = Good morning world! | length = 19 | capacity = 35
New capacity = 75
New buffer = Good morni | new length = 10
$
```

String builder example 2 (1)

```java
public class StringBuilderExample2 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s%n", sb.toString());
    char[] charArray = new char[7];
    sb.getChars(5, 12, charArray, 0);
    System.out.printf("Char array = ");
    for (char character : charArray) {System.out.print(character);}
    sb.setCharAt(5,'M');
    sb.setCharAt(13,'W');
    System.out.printf("%nNew Buffer = %s%n", sb.toString());
    sb.deleteCharAt(sb.length()-1);
    sb.append(" Again!");
    System.out.printf("New Buffer = %s%n", sb.toString());
  }
}
```

String builder example 2 (2)

```java
public class StringBuilderExample2 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s%n", sb.toString());
    char[] charArray = new char[7];
    sb.getChars(5, 12, charArray, 0);
    System.out.printf("Char array = ");
    for (char character : charArray) {System.out.print(character);}
    sb.setCharAt(5,'M');
    sb.setCharAt(13,'W');
    System.out.printf("%nNew Buffer = %s%n", sb.toString());
    sb.deleteCharAt(sb.length()-1);
    sb.append(" Again!");
    System.out.printf("New Buffer = %s%n", sb.toString());
  }
}
```

```
$ java StringBuilderExample2
Buffer = Good morning world!
```

String builder example 2 (3)

```java
public class StringBuilderExample2 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s%n", sb.toString());
    char[] charArray = new char[7];
    sb.getChars(5, 12, charArray, 0);
    System.out.printf("Char array = ");
    for (char character : charArray) {System.out.print(character);}
    sb.setCharAt(5,'M');
    sb.setCharAt(13,'W');
    System.out.printf("%nNew Buffer = %s%n", sb.toString());
    sb.deleteCharAt(sb.length()-1);
    sb.append(" Again!");
    System.out.printf("New Buffer = %s%n", sb.toString());
  }
}
```

```
$ java StringBuilderExample2
Buffer = Good morning world!
Char array = morning
```

String builder example 2 (4)

```java
public class StringBuilderExample2 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s%n", sb.toString());
    char[] charArray = new char[7];
    sb.getChars(5, 12, charArray, 0);
    System.out.printf("Char array = ");
    for (char character : charArray) {System.out.print(character);}
    sb.setCharAt(5,'M');
    sb.setCharAt(13,'W');
    System.out.printf("%nNew Buffer = %s%n", sb.toString());
    sb.deleteCharAt(sb.length()-1);
    sb.append(" Again!");
    System.out.printf("New Buffer = %s%n", sb.toString());
  }
}
```

```
$ java StringBuilderExample2
Buffer = Good morning world!
Char array = morning
New Buffer = Good Morning World!
```

String builder example 2 (5)

```java
public class StringBuilderExample2 {
  public static void main(String[] args){
    StringBuilder sb = new StringBuilder("Good morning world!");
    System.out.printf("Buffer = %s%n", sb.toString());
    char[] charArray = new char[7];
    sb.getChars(5, 12, charArray, 0);
    System.out.printf("Char array = ");
    for (char character : charArray) {System.out.print(character);}
    sb.setCharAt(5,'M');
    sb.setCharAt(13,'W');
    System.out.printf("%nNew Buffer = %s%n", sb.toString());
    sb.deleteCharAt(sb.length()-1);
    sb.append(" Again!");
    System.out.printf("New Buffer = %s%n", sb.toString());
  }
}
```

```
$ java StringBuilderExample2
Buffer = Good morning world!
Char array = morning
New Buffer = Good Morning World!
New Buffer = Good Morning World Again!
```

Class StringBuffer

- String builders created using `StringBuilder` class *are not thread safe*: if multiple threads require access to the same dynamic (i.e., modifiable) string content, use class **`StringBuffer`** instead of `StringBuilder`
    - Both classes `StringBuilder` and `StringBuffer` provide identical capabilities, but only `StringBuffer` is thread safe (i.e., synchronized)
    - If you do *not* need to access the same string builder from multiple threads, `StringBuilder` class works faster and more efficiently than `StringBuffer`

Wrapper classes

- In some situations, you need to treat primitive type values as objects (i.e., reference type values)
  - Java provides wrapper classes **`Boolean`**, **`Character`**, **`Double`**, **`Float`**, **`Byte`**, **`Short`**, and **`Integer`** for primitive types `boolean`, `char`, `double`, `float`, `byte`, `short` and `int`, respectively
  - The recommended way to covert primitive types to wrapper class objects is to use each class's static method `valueOf()`, for example:

```
int iPrim = 1; Integer i = Integer.valueOf(iPrim);
```

  - Wrapper classes can also be initialised directly by using literals (*autoboxing*):

```
Character c = 'A'; Integer i = 5;
```

Methods of class Character

- Methods of class `Character` can be useful for testing and manipulating individual character values
  - Each method takes at least a character as input argument
  - Examples of methods for *testing* characters include: **`isDefined()`**, **`isDigit()`**, **`isJavaIdentifierStart()`**, **`isJavaIdentifierPart()`**, **`isLetter()`**, **`isLetterOrDigit()`**, **`isLowerCase()`**, and **`isUpperCase()`**
  - Example of methods for manipulating characters include: **`toUpperCase()`**, returning an uppercase version of the character, and **`toLowerCase()`**, returning a lowercase version of the character

Character example (1)

```java
public class CharacterExample {
  public static void main(String[] args){
    char c = 'a';
    Character c1 = 'A';
    Character c2 = Character.valueOf(c);
    System.out.printf("c1 = %c | c2 = %s%n", c1, c2.toString());
    System.out.printf("c1 and c2 are equal? %b%n", c1.equals(c2));
    System.out.printf("c1 and c2 are equal when case ignored? %b%n",
                      c1.toString().equalsIgnoreCase(c2.toString()));
    System.out.printf("'%c' is digit? %b%n", c1, Character.isDigit(c1));
    System.out.printf("'%c' is letter? %b%n", c1, Character.isLetter(c1));
    System.out.printf("'%c' is uppercase? %b%n", c1, Character.isUpperCase(c1));
    System.out.printf("'%c' is digit? %b%n", c2, Character.isUpperCase(c2));
    System.out.printf("'%c' in uppercase is %c%n", c1, Character.toUpperCase(c1));
    System.out.printf("'%c' in uppercase is %c%n", c2, Character.toUpperCase(c2));
  }
}
```

Character example (2)

```java
public class CharacterExample {
  public static void main(String[] args){
    char c = 'a';
    Character c1 = 'A';
    Character c2 = Character.valueOf(c);
    System.out.printf("c1 = %c | c2 = %s%n", c1, c2.toString());
    System.out.printf("c1 and c2 are equal? %b%n", c1.equals(c2));
    System.out.printf("c1 and c2 are equal when case ignored? %b%n",
                      c1.toString().equalsIgnoreCase(c2.toString()));
    System.out.printf("'%c' is digit? %b%n", c1, Character.isDigit(c1));
    System.out.printf("'%c' is letter? %b%n", c1, Character.isLetter(c1));
    System.out.printf("'%c' is uppercase? %b%n", c1, Character.isUpperCase(c1));
    System.out.printf("'%c' is digit? %b%n", c2, Character.isUpperCase(c2));
    System.out.printf("'%c' in uppercase is %c%n", c1, Character.toUpperCase(c1));
    System.out.printf("'%c' in uppercase is %c%n", c2, Character.toUpperCase(c2));
  }
}
```

Different initialisations

UNIVERSITY OF ABERDEEN

Character example (3)

```java
public class CharacterExample {
  public static void main(String[] args){
    char c = 'a';
    Character c1 = 'A';
    Character c2 = Character.valueOf(c);
    System.out.printf("c1 = %c | c2 = %s%n", c1, c2.toString());
    System.out.printf("c1 and c2 are equal? %b%n", c1.equals(c2));
    System.out.printf("c1 and c2 are equal when case ignored? %b%n",
                      c1.toString().equalsIgnoreCase(c2.toString()));
    System.out.printf("'%c' is digit? %b%n", c1, Character.isDigit(c1));
    System.out.printf("'%c' is letter? %b%n", c1, Character.isLetter(c1));
    System.out.printf("'%c' is uppercase? %b%n", c1, Charac
    System.out.printf("'%c' is digit? %b%n", c2, Character.
    System.out.printf("'%c' in uppercase is %c%n", c1, Char
    System.out.printf("'%c' in uppercase is %c%n", c2, Char
  }
}
```

```
$ java CharacterExample
c1 = A | c2 = a
c1 and c2 are equal? false
c1 and c2 are equal when case ignored? true
```

Different comparisons; note that `equalsIgnoreCase()` is defined for `String`, but not `Character`

Character example (4)

```java
public class CharacterExample {
  public static void main(String[] args){
    char c = 'a';
    Character c1 = 'A';
    Character c2 = Character.valueOf(c);
    System.out.printf("c1 = %c | c2 = %s%n", c1, c2.toString());
    System.out.printf("c1 and c2 are equal? %b%n", c1.equals(c2));
    System.out.printf("c1 and c2 are equal when case ignored? %b%n",
                      c1.toString().equalsIgnoreCase(c2.toString()));
    System.out.printf("'%c' is digit? %b%n", c1, Character.isDigit(c1));
    System.out.printf("'%c' is letter? %b%n", c1, Character.isLetter(c1));
    System.out.printf("'%c' is uppercase? %b%n", c1, Character.isUpperCase(c1));
    System.out.printf("'%c' is digit? %b%n", c2, Character.isUpperCase(c2));
    System.out.printf("'%c' in uppercase is %c%n", c1, Character.toUpperCase(c1));
    System.out.printf("'%c' in uppercase is %c%n", c2, Character.toUpperCase(c2));
  }
}
```

Basic tests

```
...
'A' digit? false
'A' letter? true
'A' uppercase? true
'a' uppercase? false
```

Character example (5)

```java
public class CharacterExample {
  public static void main(String[] args){
    char c = 'a';
    Character c1 = 'A';
    Character c2 = Character.valueOf(c);
    System.out.printf("c1 = %c | c2 = %s%n", c1, c2.toString());
    System.out.printf("c1 and c2 are equal? %b%n", c1.equals(c2));
    System.out.printf("c1 and c2 are equal when case ignored? %b%n",
                      c1.toString().equalsIgnoreCase(c2.toString()));
    System.out.printf("'%c' is digit? %b%n", c1, Character.isDigit(c1));
    System.out.printf("'%c' is letter? %b%n", c1, Character.isLetter(c1));
    System.out.printf("'%c' is uppercase? %b%n", c1, Character.isUpperCase(c1));
    System.out.printf("'%c' is digit? %b%n", c2, Character.isUpperCase(c2));
    System.out.printf("'%c' in uppercase is %c%n", c1, Character.toUpperCase(c1));
    System.out.printf("'%c' in uppercase is %c%n", c2, Character.toUpperCase(c2));
  }
}
```

Basic character manipulation

```
...
'A' in uppercase is A
'a' in uppercase is A
$
```

Questions, comments?

1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 11, Session 3: Regular expressions

Regular expressions (regex)

- A *regular expression* (*regex*) is a string that describes a search pattern for matching characters in other strings
  - Such expressions are useful for validating input and ensuring that data is in a particular format
- Regular expressions can be used to perform all types of text search and text replace operations
- A large and complex regular expression is used for example to validate the syntax of a program
  - If the program code does not match the regular expression, the compiler knows that there is a syntax error in the code

Regular expression characters

- A regex consists of *literal characters* and *metacharacters*
  - Literal characters are regular characters with a literal meaning: for example, character `‘b’` is a literal character matching with character `‘b’`
  - Metacharacters are characters that have special meaning in regex: for example, metacharacter `‘.’` (dot) matches with any character
  - Some metacharacters are preceded by the *escape sequence* (backslash) `‘\’`: for example, metacharacter `‘\d’` matches with any digit
  - Backslash is also used to distinguish literal characters from metacharacters: for example, `‘*’` is a metacharacter, and `‘\*’` is a literal character matching with character `‘*’` (asterisk)

Some common metacharacters

Metacharacter	Description
.	Matches any character (except newline)
^	Matches the starting position within the string
$	Matches the ending position of the string
*	Matches the preceding element zero or more times
?	Matches the preceding element zero or one time
+	Matches the preceding element one or more times
|	Matches any of the patterns separated by '|'

Examples of using metacharacters

Regex	Example matches and non-matches
`bo.`	Matches “box” and “boy” but not “but” or “bo”
`^cat`	Matches “cat” but not “a cat”
`hat$`	Matches “hat” and “chat” but not “hatch”
`c*at`	Matches “at” and “cat” and “ccat” but not “chat”
`c?at`	Matches “at” and “cat” but not “ccat”
`c?at`	Matches “cat” and “ccat” but not “at”
`cat|dog`	Matches “cat” and “dog” but not “cow”

1495 UNIVERSITY OF ABERDEEN

Some common character classes

- The *character class* is the most basic regex concept after literal match
  - Character classes are defined by metacharacters that match with specific types of characters, like digits or whitespaces
- Some commonly used examples of character classes:

Character	Matches	Character	Matches
`\d`	Any digit	`\D`	Any non-digit
`\w`	Any word character	`\W`	Any non-word character
`\s`	Any whitespace character	`\S`	Any non-whitespace character
`\b`	Word boundaries		

Using brackets in regex

- Brackets `[ ]` are used to match any single character that is contained within the brackets
  - For example, `[abc]` matches `‘a’`, `‘b’`, and `‘c’` but not `‘d’`
- Within brackets, metacharacter `‘^’` is used to match a character that is NOT contained within the brackets
  - For example, `[^ab]` matches `‘c’` and `‘z’` but not `‘a’` or `‘b’`
- Within brackets, `‘-’` is used to define to match a range of characters
  - For example, `[a-d]` matches `‘a’`, `‘b’`, `‘c’`, and `‘d’` but not `‘e’`

Quantifiers

- Regex *quantifiers* are used to specify length of a sequence to match

Quantifier	Description
`n{x}`	Matches any string that contains a sequence of *x* times character ‘n’ (*x* is a number)
`n{x,y}`	Matches any string that contains a sequence of at least *x* but no more than *y* times character ‘n’
`n{x,}`	Matches any string that contains a sequence of at least *x* times character ‘n’

String methods for regex operations

- Class `String` provides several methods for performing regex operations
  - Method **`matches()`** takes a `String` object containing a regex as input argument and returns `true` only if the *whole* string matches the regex
  - Method `split()` uses regex expression as input to find delimiters for tokenising the string
  - Method **`replaceAll()`** uses the regex input argument to find matching substrings and replaces them with the replacement argument
  - Method **`replaceFirst()`** is similar to `replaceAll()`, but replaces only the first matching substring
  - Note that `String` method `replace()` does not support regex!

1495 UNIVERSITY OF ABERDEEN

Regex example using String methods (1)

```
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your email is not valid!");
      }
    } while(true);
  }
}
```

Regex example using String methods (2)

```java
import java.util.Scanner;
public class StringRegexE
  public static void main
    Scanner scanner = new
    System.out.println("E
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your email is not valid!");
      }
    } while(true);
  }
}
```

For simplicity, we assume that the email format is `username@domain.xxx`, and only lowercase letters are allowed in the username and URL

Regex example using String methods (3)

```java
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.printl
      }
    } while(true);
  }
}
```

Matches a sequence of one or more lowercase letters

Regex example using String methods (4)

```java
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your email is not valid!");
      }
    } while(true);
  }
}
```

Matches '@'

Regex example using String methods (5)

```java
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your em
      }
    } while(true);
  }
}
```

Matches a sequence of one or more lowercase letters

Regex example using String methods (6)

```java
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.printl
      }
    } while(true);
  }
}
```

Matches ‘.’: note that in a Java string, regex character ‘\.’ must be written as ‘\\.’, because Java compiler assumes backslash as an escape character before regex compiler!

Regex example using String methods (7)

```java
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your email is not va
      }
    } while(true);
  }
}
```

Matches a sequence of two to three lowercase letters

Regex example using String methods (8)

```
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter the email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your email is not valid!");
      }
    } while(true);
  }
}
```

Accepts different ways to write “stop”

UNIVERSITY OF ABERDEEN

Regex example using String methods (9)

```java
import java.util.Scanner;
public class StringRegexExample {
  public static void main(String[] args){
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter 'stop' when you want to finish!");
    do {
      System.out.print("Enter your email: ");
      Sinput = scanner.nextLine();
      if(input.matches("[a-z]+@[a-z]+\\.[a-z]{2,3}")) {
        System.out.println("Your email is valid!");
      }
      else if(input.matches("stop|Stop|STOP")) {
        break;
      }
      else {
        System.out.println("Your emai
      }
    } while(true);
  }
}
```

```
$ java StringRegexExample
Enter 'stop' when you want to finish!
Enter the email: teacher@school.edu
Your email is valid!
Enter the email: james.smith@company.com
Your email is not valid!
Enter the email: STOP
$
```

UNIVERSITY OF ABERDEEN

Classes Pattern and Matcher

- Java does not have any built-in regex class, but we can import **`java.util.regex`** package to work with regular expressions using the following classes:
    - Class **`Pattern`**: defines a pattern (to be used in a search)
    - Class **`Matcher`**: used to search for the pattern
    - Class **`PatternSyntaxException`**: defines an exception thrown when there is a syntax error in a regex string

Using Pattern and Matcher

- A Pattern object is created by static method **`Pattern.compile()`**
  - The first argument is a regex string specifying the pattern to be searched
  - The second argument (optional) specifies flags instructing how the search is performed, for example flag **`Pattern.CASE_INSENSITIVE`** instructs ignoring the case
- Method **`matcher()`** of the `Pattern` object is used to search the pattern in the string given as input argument; the method returns a `Matcher` object with information about the result
- Method **`find()`** of the `Matcher` object returns `true` if the pattern was found, and `false` if it was not found

Pattern and Matcher example (1)

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternMatcherExample {
  public static void main(String[] args){
    Pattern pattern = Pattern.compile("[0-3]\\d/[0-1]\\d/\\d\\d\\d\\d");
    String text = "John Smith was born on 14/05/1973.\n" +
                  "His wife Jane was born on 09/12/1976.\n" +
                  "They had a son, born on 31/10/1997 " +
                  "and a daughter, born on 01/02/2001.";
    Matcher matcher = pattern.matcher(text);
    while(matcher.find()) {
      System.out.println("Date found: " + matcher.group());
    }
  }
}
```

Pattern and Matcher example (2)

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternMatcherExample {
  public static void main(String[] args){
    Pattern pattern = Pattern.compile("[0-3]\\d/[0-1]\\d/\\d\\d\\d\\d");
    String text = "John Smith was born on 14/05/1973.\n" +
                  "His wife Jane was born on 09/12/1976.\n" +
                  "They had a son, born on 31/
                  "and a daughter, born on 01/
    Matcher matcher = pattern.matcher(text);
    while(matcher.find()) {
      System.out.println("Date found: " + matc
    }
  }
}
```

Compile pattern matcher for dates in format dd/mm/yyyy; (note that this regex validates the dates only weakly)

Pattern and Matcher example (3)

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternMatcherExample {
  public static void main(String[] args){
    Pattern pattern = Pattern.compile("[0-3]\\d/[0-1]\\d/\\d\\d\\d\\d");
    String text = "John Smith was born on 14/05/1973.\n" +
                  "His wife Jane was born on 09/12/1976.\n" +
                  "They had a son, born on 31/10/1997 " +
                  "and a daughter, born on 01/02/2001.";
    Matcher matcher = pattern.matcher(text);
    while(matcher.find()) {
      System.out.println("Date found: " + matcher.group());
    }
  }
}
```

Try to find the specified patterns in the text

UNIVERSITY OF ABERDEEN

Pattern and Matcher example (4)

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternMatcherExample {
  public static void main(String[] args){
    Pattern pattern = Pattern.compile("[0-3]\\d/[0-1]\\d/\\d\\d\\d\\d");
    String text = "John Smith was born on 14/05/1973.\n" +
                  "His wife Jane was born on 09/12/1976.\n" +
                  "They had a son, born on 31/10/1997 " +
                  "and a daughter, born on 01/02/2001.";
    Matcher matcher = pattern.matcher(text);
     while(matcher.find()) {
       System.out.println("Date found: " + matcher.group());
     }
  }
}
```

Loop through all the matching substrings found in the input string

Pattern and Matcher example (5)

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PatternMatcherExample {
  public static void main(String[] args){
    Pattern pattern = Pattern.compile("[0-3]\\d/[0-1]\\d/\\d\\d\\d\\d");
    String text = "John Smith was born on 14/05/1973.\n" +
                  "His wife Jane was born on 09/12/1976.\n" +
                  "They had a son, born on 31/10/1997 " +
                  "and a daughter, born on 01/02/2001.";
    Matcher matcher = pattern.matcher(text);
    while(matcher.find()) {
      System.out.println("Date found: " + matcher.group());
    }
  }
}
```

```
$ java PatternMatcherExample
Date found: 14/05/1973
Date found: 09/12/1976
Date found: 31/10/1997
Date found: 01/02/2001
$
```

Questions, comments?

JC2002 Java Programming

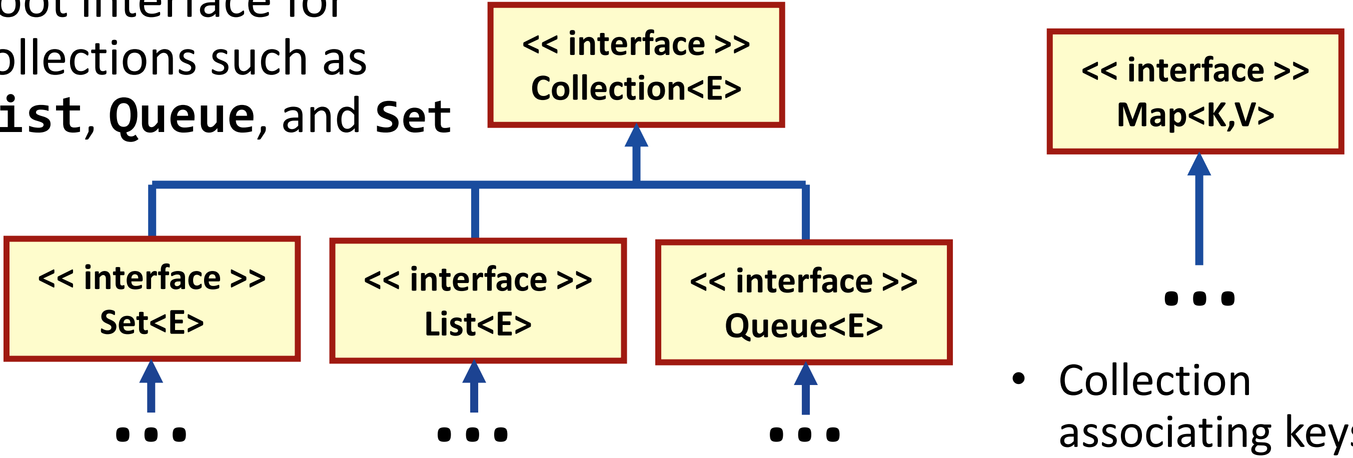
Day 11, Session 4: The basics of collections

Collections

- Any group of individual objects, represented as a single unit, is known as a *collection* of objects
  - For example, a zoo could be defined as a collection of animals
- In Java, a separate framework (*collection framework*) for handling data structures for collections has been defined
  - The main classes and interfaces for collections are included in packages **`java.util.Collection`** and **`java.util.Map`**

Collection interfaces

- Root interface for collections such as **`List`**, **`Queue`**, and **`Set`**



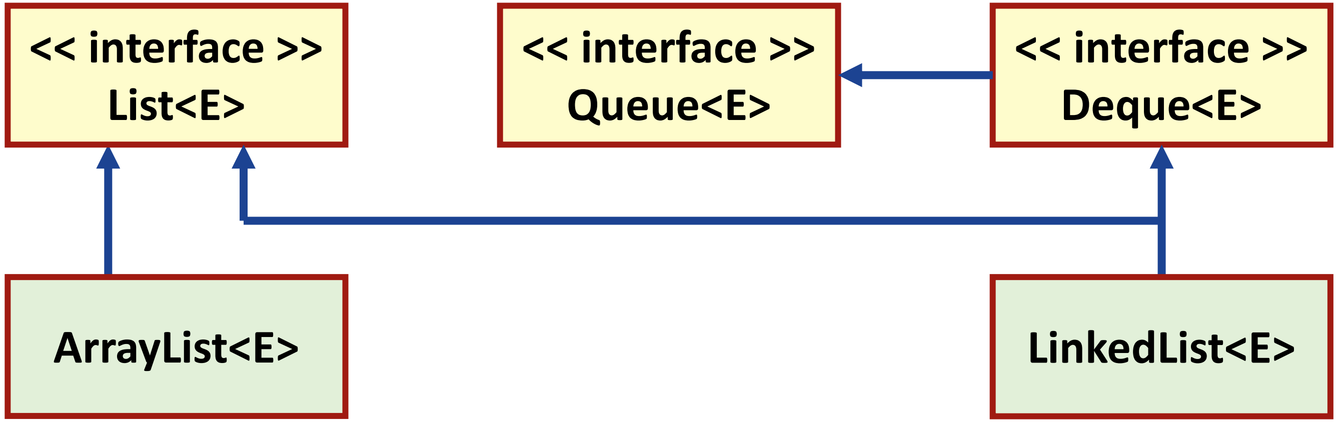
- Collection that ***cannot*** contain duplicates
- Ordered collection that may contain duplicates
- *First-in, first-out* collection modeling *waiting line*
- Collection associating keys to values, cannot contain duplicate keys, not derived from `Collection`!

Collections vs. arrays

- Unlike arrays, collections *can*:
    - Store homogeneous and heterogenous data types
    - Grow in size
    - Perform slower than arrays
- Unlike arrays, collections *cannot*:
    - Store primitive types (*int*, *char* etc.)
- Collections have many support methods defined and are better option when it comes to memory space usage

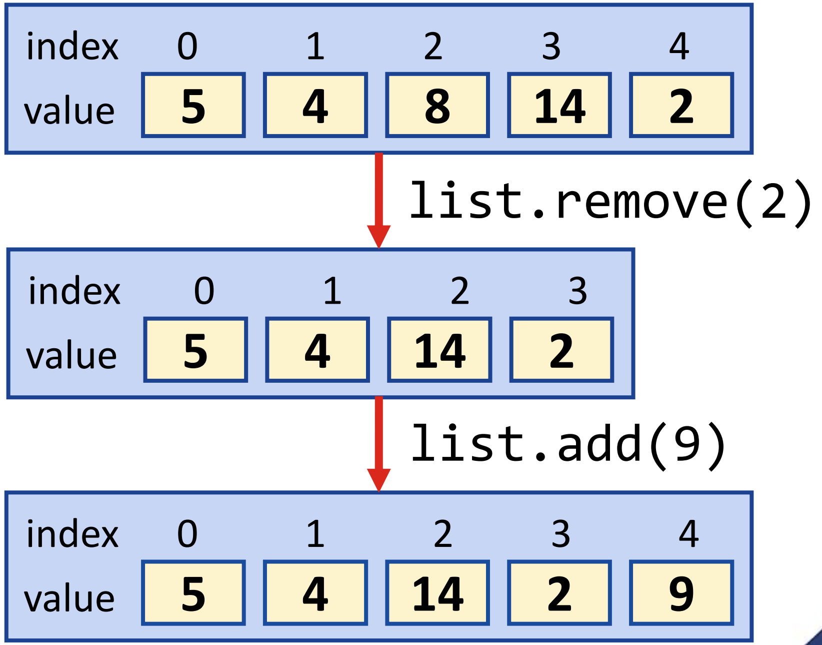
List

- There are two types of list classes: **`ArrayList`** and **`LinkedList`**
- When iterating over `List`, the order of elements is preserved



ArrayList

- Usually, the best choice of list implementations
- Access elements by calling **list.get(**index**)**
- Remove elements by calling **list.remove(**index**)**
- Add elements by calling **list.add(**data**)**



ArrayList example

```java
import java.util.* ;
public class ArrayListExample {
  public static void main ( String[] args) {
    List<String> nameList = new ArrayList<String>();
    System.out.println("initial size: " + nameList.size());
    nameList.add("Bob");      nameList.add("Cecilia");
    nameList.add("Alice");    nameList.add("Daniel");
    System.out.println("new size: " + nameList.size());
    nameList.remove(1);
    System.out.println("new size: " + nameList.size());
    nameList.add("Edward");
    System.out.println("Final name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

We refer to `ArrayList` via `List` interface to allow more flexibility: if we later notice that **`LinkedList`** is more suitable, it is easy to change.

ArrayList example: output

```java
import java.util.* ;
public class ArrayListExample {
  public static void main ( String[] args) {
    List<String> nameList = new ArrayList<String>();
    System.out.println("initial size: " + nameList.size());
    nameList.add("Bob");      nameList.add("Cecilia");
    nameList.add("Alice");    nameList.add("Daniel");
    System.out.println("new size: " + nameList.size());
    nameList.remove(1);
    System.out.println("new size: " + nameList.size());
    nameList.add("Edward");
    System.out.println("Final name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

```
$ java ArrayListExample
initial size: 0
new size: 4
new size: 3
Final name list:
Bob Alice Daniel Edward
$
```

1495 UNIVERSITY OF ABERDEEN

Iterators

- An **`Iterator`** class object can be used to loop through collections
  - "Iterating" is a technical term for looping
  - Method **`iterator()`** can be used to get an `Iterator` object for any collection
  - Methods **`hasNext()`** and **`next()`** of the iterator can be used to loop through the collection
- Note that the <u>iterator becomes invalid immediately, if the collection is modified using one of its methods</u>
  - Then, using the iterator throws `ConcurrentModificationException`
  - Helps to avoid two threads to modify collections simultaneously

Iterator example

```java
import java.util.* ;
public class IteratorExample {
  public static void main ( String[] args) {
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("Bob");    nameList.add("Cecilia");
    nameList.add("Alice");  nameList.add("Daniel");
    Iterator<String> iterator = nameList.iterator();
    while(iterator.hasNext()) {
      System.out.println(iterator.next());
      // nameList.remove(0);
      // iterator.remove();
    }
    System.out.println("Size: " + nameList.size());
  }
}
```

This will NOT work

This will work

UNIVERSITY OF ABERDEEN

Iterator example: output (1)

```java
import java.util.* ;
public class IteratorExample {
  public static void main ( String[] args) {
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("Bob");    nameList.add("Cecilia");
    nameList.add("Alice");  nameList.add("Daniel");
    Iterator<String> iterator = nameList.iterator();
    while(iterator.hasNext()) {
      System.out.println(iterator.next());
      // nameList.remove(0);
      // iterator.remove();
    }
    System.out.println("Size: " + nameList.size());
  }
}
```

```
$ java IteratorExample
Bob
Cecilia
Alice
Daniel
Size: 4
$
```

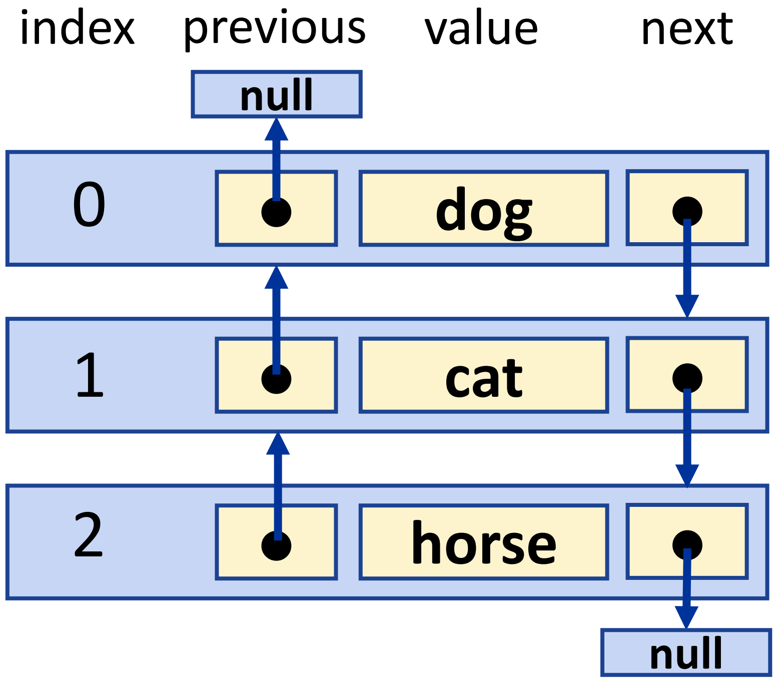
Iterator example: output (2)

```java
import java.util.* ;
public class IteratorExample {
  public static void main ( String[] args) {
    ArrayList<String> nameList = new ArrayList<String>();
    nameList.add("Bob");    nameList.add("Cecilia");
    nameList.add("Alice");  nameList.add("Daniel");
    Iterator<String> iterator = nameList.iterator();
    while(iterator.hasNext()) {
      System.out.println(iterator.next());
      // nameList.remove(0);
      iterator.remove();
    }
    System.out.println("Size: " + nameList.size());
  }
}
```

```
$ java IteratorExample
Bob
Cecilia
Alice
Daniel
Size: 0
$
```

LinkedList

- Start with *head*, ends with *tail*
- Elements are linked to the next element
- Efficient insertion and deletion, but bad for memory
- Access elements by calling **`list.get(`**`index`**`)`**
- Add elements by calling **`list.add(`**`data`**`)`** or **`list.push(`**`data`**`)`**



LinkedList example

```java
import java.util.* ;
public class LinkedListExample {
  public static void main ( String[] args) {
    LinkedList<String> nameList = new LinkedList<String>();
    nameList.push("Bob");
    nameList.add("Daniel");
    nameList.addFirst("Alice");
    nameList.add(2, "Cecilia");
    System.out.println("Name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

Adds in the beginning of the list

Adds in the end of the list

LinkedList example: output

```java
import java.util.* ;
public class LinkedListExample {
  public static void main ( String[] args) {
    LinkedList<String> nameList = new LinkedList<String>();
    nameList.push("Bob");
    nameList.add("Daniel");
    nameList.addFirst("Alice");
    nameList.add(2, "Cecilia");
    System.out.println("Name list:");
    for(int i=0; i<nameList.size(); i++) {
      System.out.print(nameList.get(i) + " ");
    }
    System.out.println("");
  }
}
```

```
$ java LinkedListExample
Name list:
Alice Bob Cecilia Daniel
$
```

ArrayList vs. LinkedList

- Basically, same things can be done with both types of lists: their main difference is the internal representation of data
  - In `ArrayList`, it is faster to find an element at certain index, because the elements are located in memory in fixed order
  - In `LinkedList`, it is faster to add and remove elements, because there is no need to move large blocks of data in memory
- The choice depends on the application
  - `LinkedList` is better if there is often need to add and remove elements
  - `ArrayList` is better if you just need to modify elements without adding or removing

Some important methods in collections

Method	Description
`sort()`	Sorts the elements of a `List`
`binarySearch()`	Locates an element of a `List` using efficient binary search algorithm
`reverse()`	Reverses the elements of a `List`
`shuffle()`	Reorders the elements of a `List` randomly
`fill()`	Sets every element in a `List` to refer to a specific object
`copy()`	Copies references from one `List` to another

Deck of cards example: class Card

```
import java.util.* ;
class Card {
  public enum Face {Ace, Two, Three, Four, Five, Six, Seven,
                    Eight, Nine, Ten, Jack, Queen, King}
  public enum Suit {Clubs, Diamonds, Spades, Hearts}
  private final Face face;
  private final Suit suit;
  public Card(Face face, Suit suit) {
    this.face = face; this.suit = suit;
  }
  public Face getFace()  { return face; }
  public Suit getSuit()  { return suit; }
  public String toString() {
    return String.format("%s of
  }
}
```

```
$ java LinkedListExample
Name list:
Alice Bob Cecilia Daniel
$
```

Deck of cards example: using List

```java
public class DeckOfCards {
  private List<Card> cards;
  public DeckOfCards() {
    Card deck[] = new Card[52];
    int count = 0;
    for(Card.Suit suit: Card.Suit.values()) {
      for(Card.Face face: Card.Face.values()) {
        deck[count++] = new Card(face, suit);
      }
    }
    cards = Arrays.asList(deck);
    Collections.shuffle(cards);
  }
  public void printCards() {
    for(int i=0; i<52; i++) {
      System.out.printf("%-19s%s", cards.get(i),
        ((i+1) % 4 == 0) ? "\n" : "");
    }
  }
```

Converts an Array to a List

```java
  public static void main ( String[] args) {
    DeckOfCards deck = new DeckOfCards();
    deck.printCards();
  }
}
```

1495 UNIVERSITY OF ABERDEEN

Deck of cards example: output

```
$ java DeckOfCards
King of Spades     Queen of Diamonds  Eight of Diamonds  Six of Hearts
Five of Hearts     Jack of Hearts     Jack of Clubs      Three of Spades
Five of Spades     Ace of Spades      Eight of Hearts    Four of Spades
King of Diamonds   Four of Hearts     Queen of Spades    Nine of Clubs
Jack of Spades     Three of Hearts    Jack of Diamonds   Four of Clubs
Four of Diamonds   Two of Clubs       Ace of Clubs       Ace of Hearts
Ten of Clubs       Ace of Diamonds    Six of Spades      Eight of Spades
Six of Clubs       Seven of Clubs     Five of Diamonds   Eight of Clubs
Nine of Spades     Seven of Hearts    Two of Spades      Two of Diamonds
King of Clubs      Nine of Diamonds   Queen of Clubs     Three of Diamonds
Nine of Hearts     Seven of Spades    Ten of Spades      Three of Clubs
Two of Hearts      Six of Diamonds    Seven of Diamonds  Ten of Diamonds
King of Hearts     Queen of Hearts    Five of Clubs      Ten of Hearts
$
```

Example of sort() and binarySearch()

```java
import java.util.* ;
public class BinarySearchExample {
  public static void main ( String[] args) {
    String[] names = {"Bob","Alice","Edward","Cecilia","David","Frank"};
    List<String> list = new ArrayList<>(Arrays.asList(names));
    Collections.sort(list);
    for(String name : list) {
      System.out.println(name);
    }
    int index = Collections.binarySearch(list, "Edward");
    System.out.printf("Index of Edward is %d\n", index);
  }
}
```

UNIVERSITY OF ABERDEEN

Output of sort() and binarySearch()

```java
import java.util.* ;
public class BinarySearchExample {
  public static void main ( String[] args) {
    String[] names = {"Bob","Alice","Edward","Cecilia","David","Frank"};
    List<String> list = new ArrayList<>(Arrays.asList(names));
    Collections.sort(list);
    for(String name : list) {
      System.out.println(name);
    }
    int index = Collections.binarySearch(list, "David");
    System.out.printf("Index of David is %d\n", index);
  }
}
```

```
$ java BinarySearchExample
Alice
Bob
Cecilia
David
Edward
Frank
Index of David is 3
$
```

Summary

- In many applications, string operations like searching for substrings, concatenating strings, and modifying string content are essential
  - In Java, `String` objects are immutable: their content cannot be changed after they are created
  - Class `StringBuilder` can be used to create dynamic string buffers
- Regular expressions are commonly used for defining string patterns for searching and validating string content in a specific format
- Java collection framework is useful for handling data structures holding a group of items, such as *lists*, *sets*, and *queues*

Questions, comments?