


Build COMPETENCY
across your TEAM



Microsoft Partner
Gold Cloud Platform
Silver Learning

Exploring Java Basics



A cluster of approximately 15 light gray squares of various sizes, some overlapping, arranged in a loose, abstract pattern in the upper right quadrant of the slide.

The Object Class
Wrapper Classes
Type casting
Using Scanner Class
String Handling
Date and Time API





The Object Class



Object class in Java

- Object class is present in java.lang package.
- Every class in Java is directly or indirectly derived from the Object class.
 - If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes.
- Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Object class in Java

There are methods in **Object** class:

- toString()
 - hashCode()
 - equals()
 - finalize()
 - clone()
 - getClass()
- The remaining three methods **wait()**, **notify()** **notifyAll()** are related to Concurrency.



Wrapper Classes



Wrapper Classes in Java

- Wrapper class is a class whose object wraps or contains a primitive data types.
- When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Need of Wrapper Classes

- They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

Primitive Data types and their Corresponding Wrapper class

Primitive Date Type	Wrapper Class
char	Character
Byte	Byte
Short	Short
Long	Long
float	Float
double	Double
Boolean	Boolean

Autoboxing and Unboxing

- **Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.
- For example – conversion of int to Integer, long to Long, double to Double etc.

```
import java.util.ArrayList;
class Autoboxing
{
    public static void main(String[] args)
    {
        char ch = 'a';

        // Autoboxing- primitive to Character object conversion
        Character a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);

        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

Autoboxing and Unboxing

- **Unboxing:** It is just the reverse process of autoboxing.
- Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.
- For example – conversion of Integer to int, Long to long, Double to double etc.

```
// Java program to demonstrate Unboxing
import java.util.ArrayList;

class Unboxing
{
    public static void main(String[] args)
    {
        Character ch = 'a';

        // unboxing - Character object to primitive conversion
        char a = ch;

        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24);

        // unboxing because get method returns an Integer object
        int num = arrayList.get(0);

        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Type Casting

Type Casting in Java

- We are aware the Java type system is made up of two kinds of types:
 - primitives
 - references.

- Assigning a value of one type to a variable of another type is known as **Type Casting**.

```
int x = 10;  
byte y = (byte)x;
```

- In Java, type casting is classified into two types,
 - Widening Casting(Implicit)
 - Narrowing Casting(Explicitly done)

byte → short → int → long → float → double
—————→
widening

double → float → long → int → short → byte
—————→
Narrowing

Widening or Automatic type conversion

- Automatic Type casting take place when,
 - the two types are compatible
 - the target type is larger than the source type

```
int i = 100;
```

```
long l = i; //no explicit type casting required
```

```
float f = l; //no explicit type casting required
```

```
System.out.println("Int value "+i);
```

```
System.out.println("Long value "+l);
```

```
System.out.println("Float value "+f);
```


Narrowing or Explicit type conversion

- When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

```
double d = 100.04;
```

```
long l = (long)d; //explicit type casting required
```

```
int i = (int)l;    //explicit type casting required
```

```
System.out.println("Double value "+d);
```

```
System.out.println("Long value "+l);
```

```
System.out.println("Int value "+i);
```

Primitive vs. Reference

- Although primitive conversions and reference variable casting may look similar, they're quite different concepts.
- In both cases, we're "turning" one type into another. But, in a simplified way, a primitive variable contains its value, and conversion of a primitive variable means irreversible changes in its value.

```
double myDouble = 1.1;
```

```
int myInt = (int) myDouble;
```

```
assertNotEquals(myDouble, myInt);
```

- After the conversion in the above example, *myInt* variable is 1, and we can't restore the previous value 1.1 from it.

Primitive vs. Reference

- Reference variables are different; the reference variable only refers to an object but doesn't contain the object itself.
- And casting a reference variable doesn't touch the object it refers to, but only labels this object in another way, expanding or narrowing opportunities to work with it. Upcasting narrows the list of methods and properties available to this object, and downcasting can extend it.
- A reference is like a remote control to an object. The remote control has more or fewer buttons depending on its type, and the object itself is stored in a heap. When we do casting, we change the type of the remote control but don't change the object itself.

Upcasting

- **Casting from a subclass to a superclass is called upcasting.**
- Typically, the upcasting is implicitly performed by the compiler.
- Upcasting is closely related to inheritance – another core concept in Java.
- It's common to use reference variables to refer to a more specific type. And every time we do this, implicit upcasting takes place.

Downcasting

- It's the casting from a superclass to a subclass.



Using Scanner Class

Java Scanner Concept

- The meaning of the Scanner class for many in the beginning is a bit complicated to understand, but with time the programmer just gets used to its definition. A simple text scanner can parse primitive types and Strings using regular expressions.
- The class Scanner aims to separate the entry of text into blocks, generating the known tokens, which are sequences of characters separated by delimiters by default corresponding to blanks, tabs, and newline.
- With this class you can convert text to primitives, and these texts can be considered as objects of type String, InputStream and files.

Scanner Class in Java

- To create an object of Scanner class, we usually pass the predefined object `System.in`, which represents the standard input stream. We may pass an object of class `File` if we want to read input from a file.
- To read numerical values of a certain data type `XYZ`, the function to use is `nextXYZ()`. For example, to read a value of type `short`, we can use `nextShort()`
- To read strings, we use `nextLine()`.
- To read a single character, we use `next().charAt(0)`. `next()` function returns the next token/word in the input as a string and `charAt(0)` function returns the first character in that string.

Scanner Class methods

- **close():** Closes the current scanner.
- **findInLine():** Attempts to find the next occurrence of the specified pattern ignoring delimiters..
- **hasNext():** Returns true if this scanner has another token in its input.
- **hasNextXYZ():** Returns true if the next token in this scanner's input can be interpreted as a XYZ in the default radix using the nextXYZ() method. Here XYZ can be any of these types: BigDecimal, BigInteger, Boolean, Byte, Short, Int, Long, Float, or Double.
- **match():** Returns the match result of the last scanning operation performed by this scanner.
- **next():** Finds and returns the next complete token from this scanner.

Scanner Class methods

- **nextXYZ():** Scans the next token of the input as a XYZ where XYZ can be any of these types: boolean, byte, short, int, long, float or double.
- **nextLine():** Advances this scanner past the current line and returns the input that was skipped.
- **radix():** Returns the current index of the Scanner object.
- **remove():** This operation is not supported by the implementation of an Iterator.
- **skip():** Skip to the next search for a specified pattern ignoring delimiters.
- **string():** Returns a string representation of the object is a Scanner.

String Handling

String Handling

- In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

```
char[] ch={'H','e','l','l','o'};  
String s=new String(ch);
```

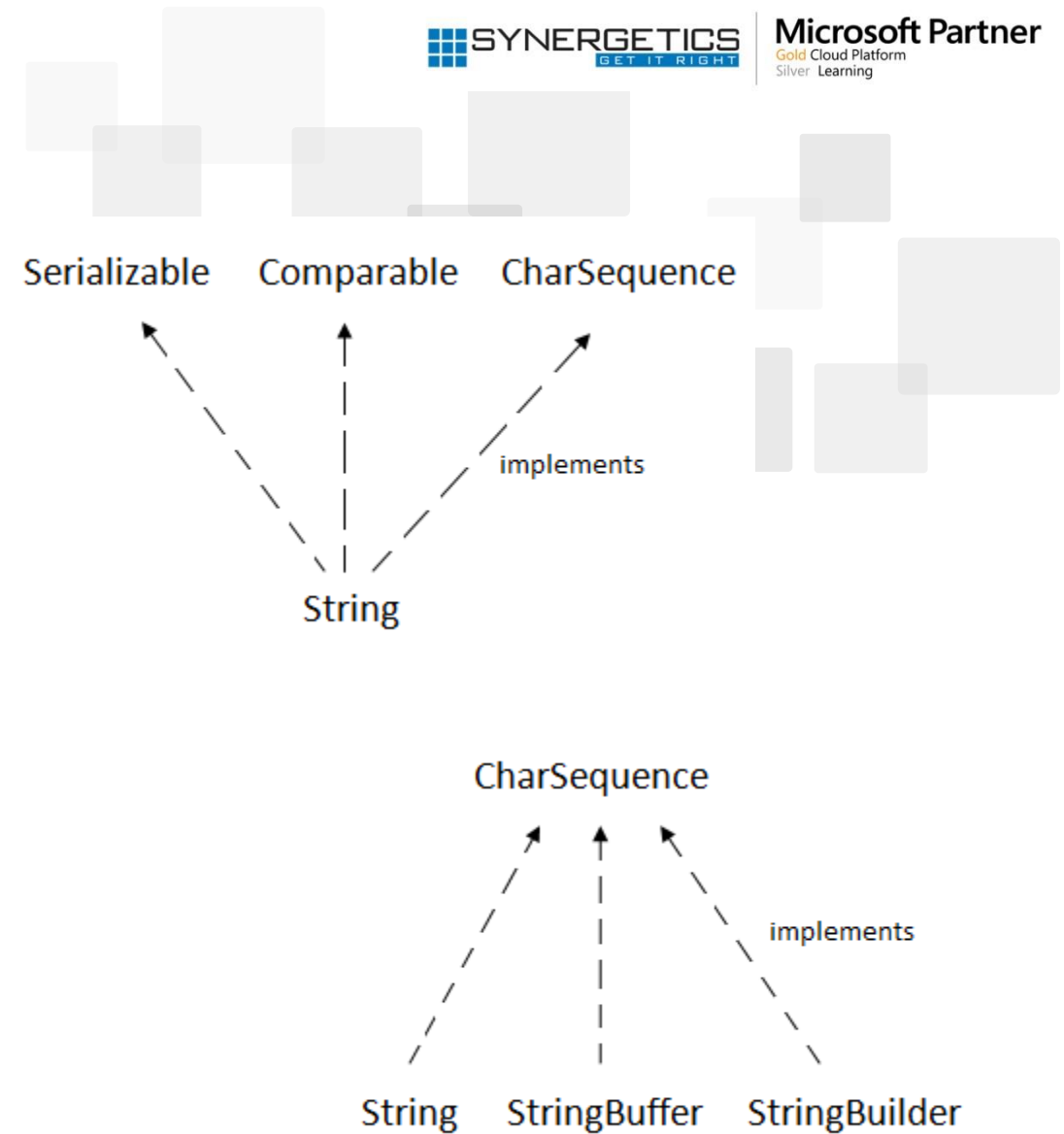
is same as:

```
String s="Hello";
```

- **Java String** class provides a lot of methods to perform operations on string such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

String Handling

- The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces →
- CharSequence Interface →
- The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.



String Handling

- The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created.
- For mutable strings, you can use StringBuffer and StringBuilder classes.

String Handling

- What is String in java

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The `java.lang.String` class is used to create a string object.

How to create a string object?

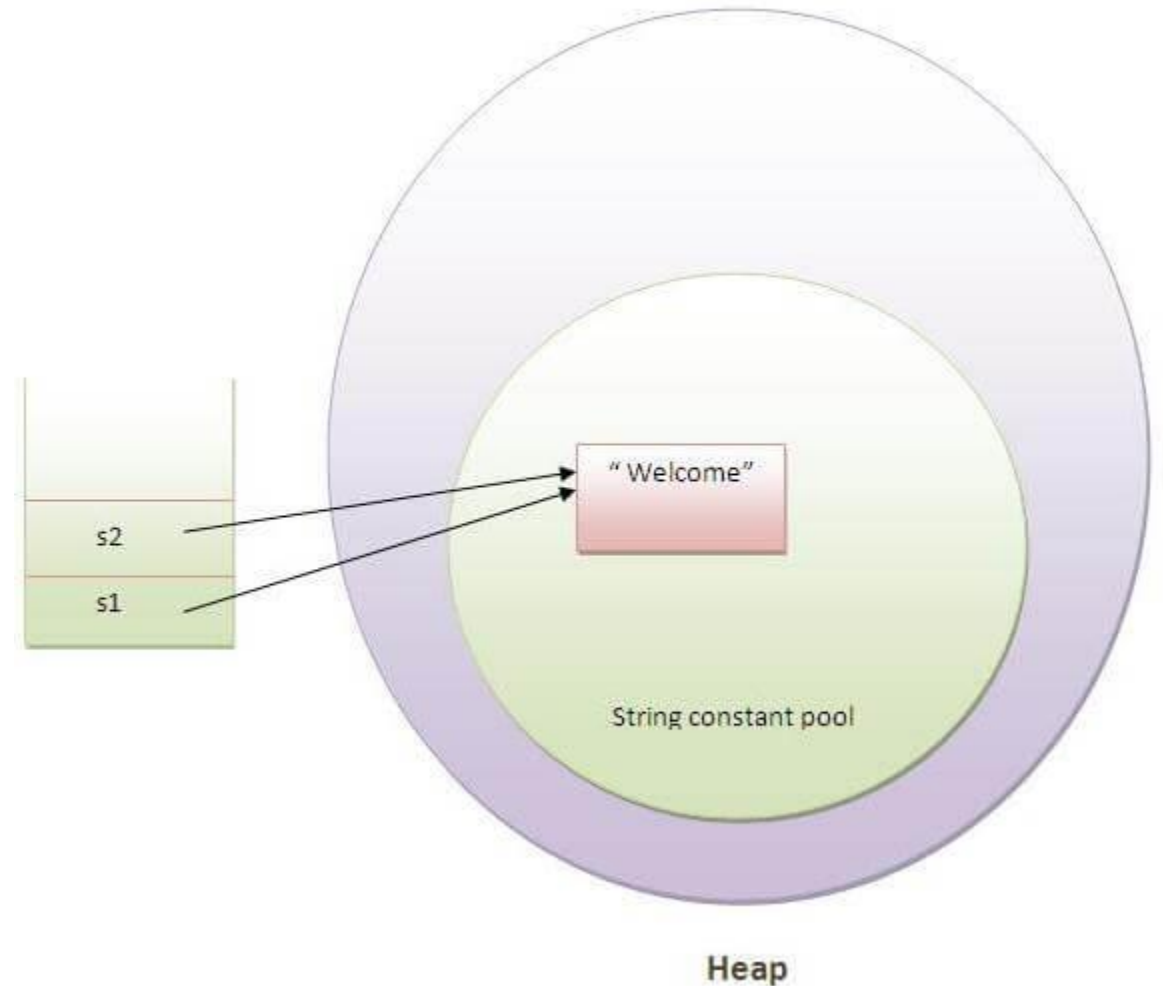
- There are two ways to create String object:
 - By string literal
 - By new keyword

By String Literal

- Java String literal is created by using double quotes. `String s="welcome";`
- Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
```

```
String s2="Welcome";//It doesn't create a new instance
```



By String Literal

- In the above example, only one object will be created.
- Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object.
- After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

By String Handling

Why Java uses the concept of String literal?

- To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

By new keyword

- String s=**new** String("Welcome");//creates two objects and one reference variable
- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.
- The variable s will refer to the object in a heap (non-pool).

Java String class methods

No	Method	Description
1	<u>char charAt(int index)</u>	returns char value for the particular index
2	<u>int length()</u>	returns string length
3	<u>static String format(String format, Object... args)</u>	returns a formatted string.
4	<u>static String format(Locale l, String format, Object... args)</u>	returns formatted string with given locale.
5	<u>String substring(int beginIndex)</u>	returns substring for given begin index.
6	<u>String substring(int beginIndex, int endIndex)</u>	returns substring for given begin index and end index.
7	<u>boolean contains(CharSequence s)</u>	returns true or false after matching the sequence of char value.
8	<u>static String join(CharSequence delimiter, CharSequence... elements)</u>	returns a joined string.
9	<u>static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)</u></u>	returns a joined string.

Java String class methods

No	Method	Description
10	<u>boolean equals(Object another)</u>	checks the equality of string with the given object.
11	<u>boolean isEmpty()</u>	checks if string is empty.
12	<u>String concat(String str)</u>	concatenates the specified string.
13	<u>String replace(char old, char new)</u>	replaces all occurrences of the specified char value.
14	<u>String replace(CharSequence old, CharSequence new)</u>	replaces all occurrences of the specified CharSequence.
15	<u>static String equalsIgnoreCase(String another)</u>	compares another string. It doesn't check case.
16	<u>String[] split(String regex)</u>	returns a split string matching regex.
17	<u>String[] split(String regex, int limit)</u>	returns a split string matching regex and limit.
18	<u>String intern()</u>	returns an interned string.
19	<u>int indexOf(int ch)</u>	returns the specified char value index.

Java String class methods

No	Method	Description
20	<u>int indexOf(int ch, int fromIndex)</u>	returns the specified char value index starting with given index.
21	<u>int indexOf(String substring)</u>	returns the specified substring index.
22	<u>int indexOf(String substring, int fromIndex)</u>	returns the specified substring index starting with given index.
23	<u>String toLowerCase()</u>	returns a string in lowercase.
24	<u>String toLowerCase(Locale l)</u>	returns a string in lowercase using specified locale.
25	<u>String toUpperCase()</u>	returns a string in uppercase.
26	<u>String toUpperCase(Locale l)</u>	returns a string in uppercase using specified locale.
27	<u>String trim()</u>	removes beginning and ending spaces of this string.
28	<u>static String valueOf(int value)</u>	converts given type into string. It is an overloaded method.



Data Time API

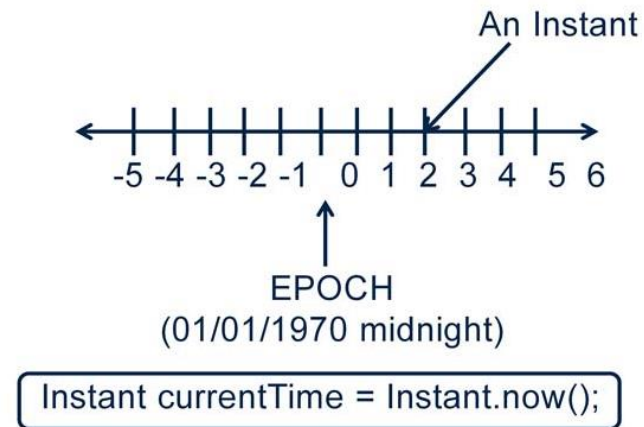


Date and Time API

- Added in Java SE 8 under java.time package.
- Enhanced API to make extremely easy to work with Date and Time.
- Immutable API to store date and time separately.
 - Instant
 - LocalDate
 - LocalTime
 - LocalDateTime
 - ZonedDateTime
- It has also added classes to measure date and time amount.
 - Duration
 - Period
- Improved way to represent units like day and months.
- Generalised parsing and formatting across all classes.

The Instance Class

- An object of instant represent point on the time line
- The reference point is the standard java epoch
- This class is useful to represent machine timestamp.



- The static method "now" of Instant class is used to represent currenttime.

The Local Date Class

- it represent date without time and zone
- Useful to represent date events like birthdate
- Following table shows important methods of LocalDate

Method	Description
now	A static method to return today's date
of	Creates local date from year, month and date
getXXX ()	Used to return various part of date
plusXXX()	Add the specified factor and return a LocalDate
minusXXX	Subtracts the specified factor and return a LocaliDate
isXXX()	Performs checks on LocalDate and returns Boolean value.
withXXX()	Returns a copy of LocalDate with the factor set to the givenvalue.

The Local Date Class

- `LocalDate now = LocalDate.now();`
- `LocalDate independence = LocalDate.of(1947, Month.AUGUST, 15);`
- `System.out.println("Independence:" + independence);`
- `System.out.println("Today:" + now);`
- `System.out.println("Tomorrow:" + now.plusDays(1));`
- `System.out.println("Last Month " + now.minusMonths(1));`
- `System.out.println("Is leap?:" + now.isLeapYear());`
- `System.out.println("Move to 38th day of month:" + now.withDayOfMonth(30));`

The ZonedDateTime Class

- It stores all date and time fields, to a precision of nanoseconds, as well as a time-zone and zone offset.
- Useful to represent arrival and departure time in airline applications.
- Following table shows important methods of ZonedDateTime:

Method	Uses
now	A static method to return today's date.
of	Overloaded static method to create zoned date time object.
getXXX ()	Used to return various part of ZonedDateTime.
plusXXX()	Add the specified factor and return a ZonedDateTime.
minusXXX()	Subtracts the specified factor and return a ZonedDateTime.
isXXX()	Performs checks on ZonedDateTime and returns Boolean value.
withXXX()	Returns ZonedDateTime with the factor set to the given value.

The ZonedDateTime Class

- `ZonedDateTime currentTime = ZonedDateTime.now();`
- `ZonedDateTime currentTimeInParis = ZonedDateTime.now(ZoneId.of("Europe/Paris"));`
- `System.out.println("India " + currentTime);`
- `System.out.println("Paris: " + currentTimeInParis);`

Period and Duration

- The Period class models a date-based amount of time, such as five days, a week or three years.
- Duration class models a quantity or amount of time in terms of seconds and nanoseconds. It is used represent amount of time between two instants.
- Following table shows important and common methods of both:

Method	Uses
between	Use to create either Period or Duration between LocalDates.
of	Creates Period/Duration based on given year, months & days.
ofXXX()	Creates Period/Duration based on specified factors.
getXXX ()	Used to return various part of Period/Duration.
plusXXX()	Add the specified factor and return a LocalDate.
minusXXX(Subtracts the specified factor and return a LocalDate.
isXXX()	Performs checks on LocalDate and returns Boolean value.
withXXX()	Returns a copy of LocalDate with the factor set to the given value.

Period and Duration

```
LocalDate start = LocalDate.of(1947, Month.AUGUST, 15);
```

```
LocalDate end = LocalDate.now(); //18/02/2015
```

```
Period period = start.until( end );
```

```
System.out.println("Days: " + period.get(ChronoUnit.DAYS));
```

```
System.out.println("Months: " + period.get(ChronoUnit.MONTHS));
```

```
System.out.println("Years: " + period.get(ChronoUnit.YEARS));
```

Formatting and Parsing Date and Time

- Java SE 8 adds `DateTimeFormatter` class which can be used to format and parse the date and time.
- To either format or parse, the first step is to create instance of `DateTimeFormatter`.
- Following are few important methods available on this to create `DateTimeFormatter`.

Method	Uses
<code>ofLocalizedDate(dateStyle)</code>	Date style formatter from locale
<code>ofLocalizedTime(timeStyle)</code>	Time style formatter from locale
<code>ofLocalizedDateTime(dateTimeStyle)</code>	Date and time style formatter from locale
<code>ofPattern(StringPattern)</code>	Custom style formatter from string

- Once formatter object created, parsing/formatting is done by using `parse()` and `format()` methods respectively. These methods are available on all major date and time classes .

Formatting and Parsing Date and Time

- To format or parse a date/time, first we need to create instance of `DateTimeFormatter`. Following example shows, how to format a date using this class. The below example shows how to format the `LocalDate` in medium style.

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.Medium);  
LocalDate currentDate = LocalDate.now();  
System.out.println(currentDate.format(formatter))
```

- The following example shows how to parse a text string into date.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");  
String text = "12/02/1982";  
LocalDate date = LocalDate.parse(text, formatter);  
System.out.println(date);
```

Q & A

Contact: amitmahadik@synergetics-india.com

Thank You

