8 week to practice: 17/2/2025-17/4/2025

# **1. Working with Java data types (10%)**

## **1.1 Operator, casting, unboxing-autoboxing**

### **1.1.1 Arithmetic Operators**

-Addition+, subtraction-, multiplication\*, division/, integer remainder %

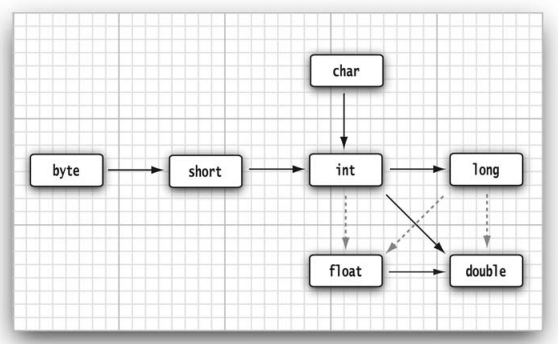
-**Note**: integer / 0 =exception, floating / 0 = NaN or infinite

### **1.1.2 Mathematical Functions and Constants**

-The Math class contains an assortment of mathematical functions.

-Almost methods in java.lang.Math return double, except: max, min, abs, round

### **1.1.3 Conversions between Numeric Types**

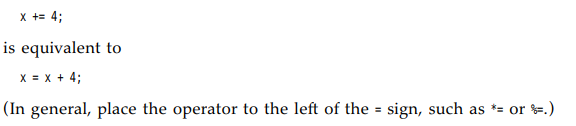


### **1.1.4 Casts**

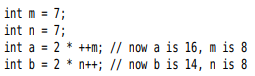
 

Note: boolean->number: 

### **1.1.5 Combining Assignment with Operators**



### **1.1.6 Increment and Decrement Operators**



### **1.1.7 Relational and boolean Operators**

- ==, !=, <, &&, ||..

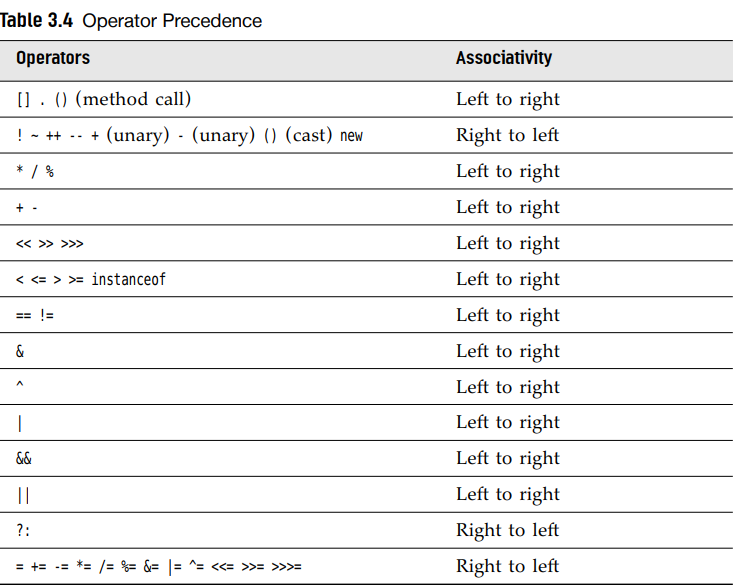
-The ternary ?: 

### **1.1.8 Bitwise Operators**

  
Note: ~ N = -(N+1)

-Shift operators: <<, >>

### **1.1.9 Parentheses and Operator Hierarchy**



## **1.2 String-StringBuilder**

### **1.2.1 String**

-Constructor: (), byte[], char[], String, StringBuilder

- Static methods: join, valueOf, format

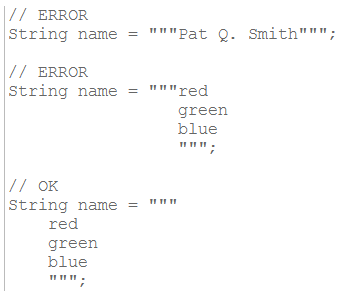
-Instatnce methods: charAt, concat, contains, startsWith, endsWith, equals, equalIgnoreCase, indexOf, indent, isBlank, isEmpty, lastIndexOf, length, repeat, replace, replaceAll, replaceFirst, split, strip, stripIndent, stripIndent, stripLeading, stripTrailing, subString, toLowerCase, toUpperCase, trim

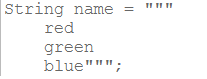
### **1.2.2 StringBuilder**

-Constructor: (), String

-Instance methods: append, capacity, charAt, delete, deleteCharAt, indexOf, lastIdexOf, insert, length(), replace, reverse, subString

### **1.2.3 Text block:**





-Trailing white space:Incidental white space + essential white space

## **1.3 Typer inference with var**

-var is not a keyword, can use var as variable name

-Var can use in:

+static/instance initialization block

+as a local variable

+iteration variable in enhanced for-loop

+as looping index in for-loop

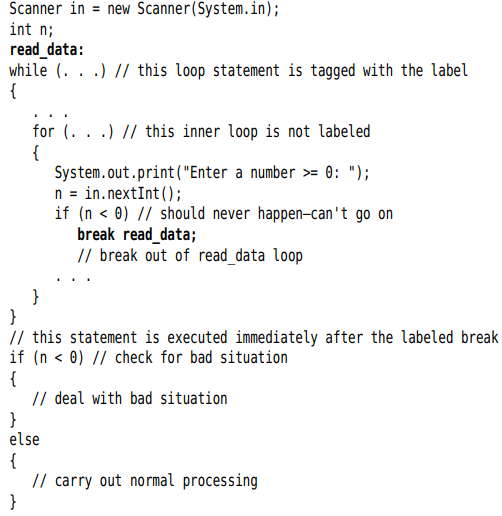
+as a return value from another method,

+as a return value in a method,

## **1.4 Controlling Program Flow (5%)**

-**break**: exit a switch and break out of a loop.

-label and break: break out of all nested loops



-**continue**: transfer control to the header of innermost enclosing loop

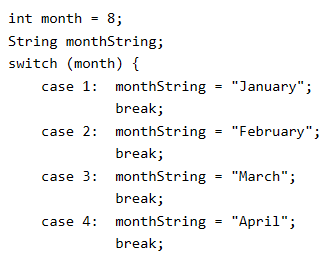
|  |  |
| --- | --- |
| |  | | --- | | -Note: labeled break or continue statement must always exist inside the loop where the label is declared | |

## **1.5 if/else, swith-case, loops**

### **1.5.1 Switch**

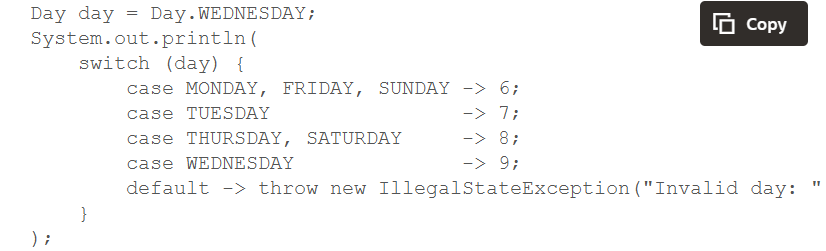
-works with: byte, short, char, int+ their wrap+ enumarted+String

-Switch statement

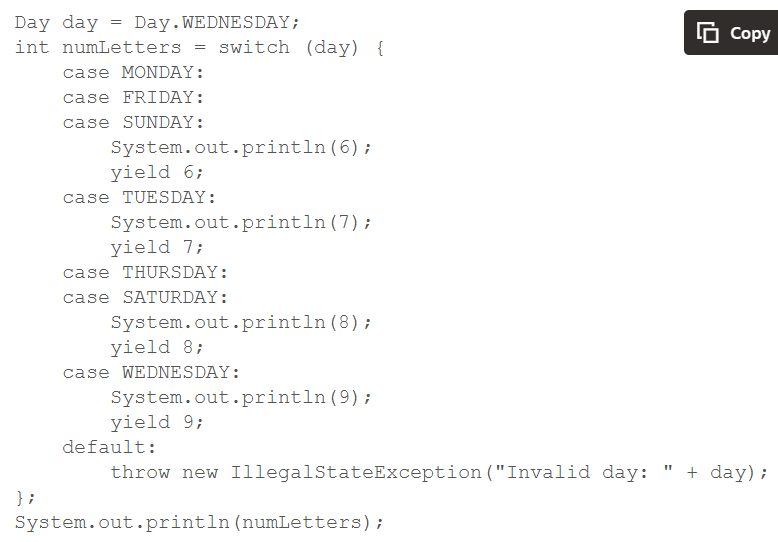


-Switch expression

+”case L->: Labels:



+”case L:” Statements and yield Statement



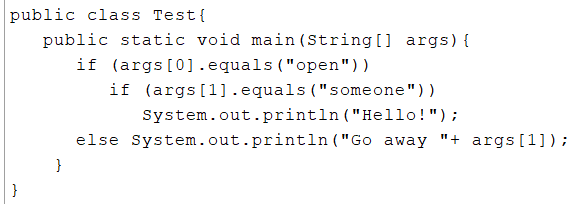
-Note:

+The default label is optional for switch statement and required for a switch expression only when the case labels are not exhaustive.

+Can’t mix switch expression+statement

### **1.5.2 If-else**

-An else clause belongs to the innermost if the first if() condition fails+there is no else associated to execute



# **2. Java Object-Oriented (30%)**

## **2.1 Class-objects declaration, initialization, life cycles**

-Default field Initialization:

If you don’t set a field explicitly in a constructor, it is automatically set to a **default value**: numbers to 0, boolean to false, object reference to null.

-Constructor with no Arguments:

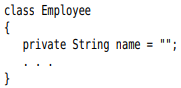
+Constructor with no arguments: creates an object whose state is set to an appropriate default.

+If a class with no constructors: a no-argument constructor is provided with all default values fields.

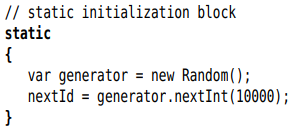
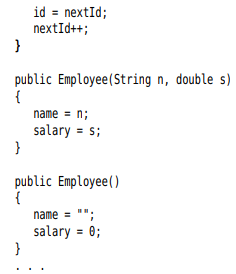
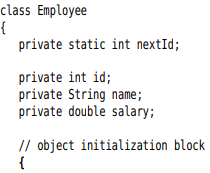
+If a class supplies at least 1 constructor but does not supply no-argument constructor => cannot construct object with no-arguments

-Explicit Field Initialization:

You can simply assign a value to any field in class definition. This assignment is carried out before the constructor executes.



-Initialization Blocks:



-Constructor flow

+First line calls a second constructor? Executes

+fields: default values + fields initializers and initialization block in the order in class declaration

+The body of constructor.

## **2.2 Fields, methods with instance, static, overloading**

-**Overloading**: several methods have the same name but different parameters and different return types.

## **2.3 Nested class, inner class, local class, anonymous class**

### **2.3.1 Inner Class to access object state**

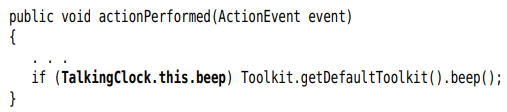
-**Can’t** have **static methods**, **can** have **final static fields**

-**Inner class method** gets to **access** both its own **data fields** and **those** of the **outer object** creating it.

### **2.3.2 Syntax rules for inner classes**

-**OuterClass.this** denotes the **outer class** reference

-**Method**:



-**Inner object constructor** more **explicitly**:







+Refer to **Inner class** when it occurs outside the scope of outer class: 

### **2.3.3 Local inner classes**

-You can define the class locally in a **single method**.

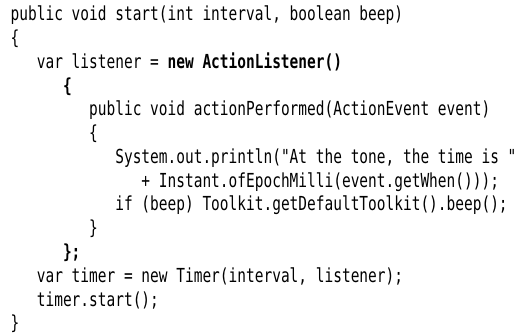
-Local classes are **not declared** with **access specifier**. Their **scope** is restricted to the **block** they’re **declared**

-**Local class** are completely **hidden** from the **outside**

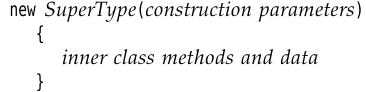
-**Local classes** can even **access** **local variables**. Those variable must be **effectively final***.*

### **2.3.4 Anonymous Inner Classes**

**Anonymous inner class**: class that don’t have name, used to make a single object of this class



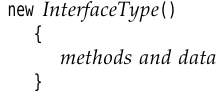
-General syntax:



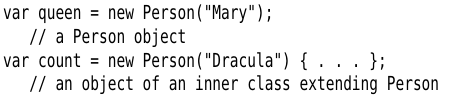
+SuperType can be an **interface, class**

-Anonymous inner class can’t have constructors. The construction parameters are given to superclass constructor.

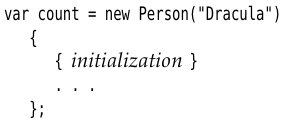
-When an inner class implements interface:



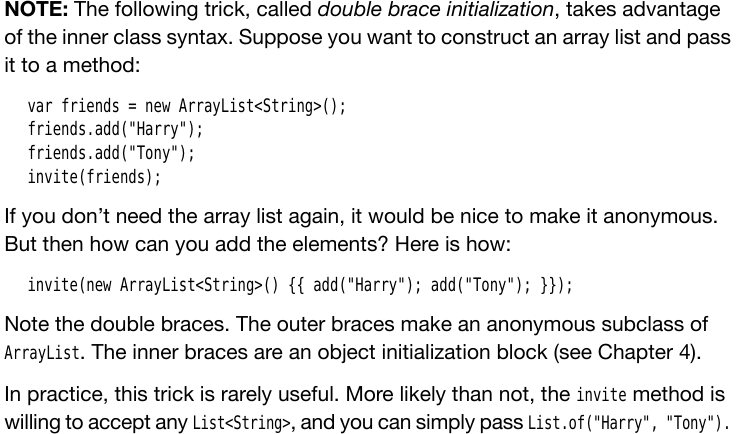
-Different:



-Note: You can provide object initialization block



-**Note**:

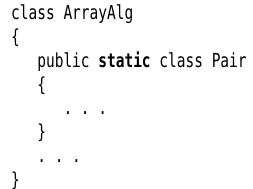


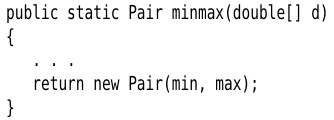
-**Anonymous class** can used in **static method.**

### **2.3.5 Static Inner Classes**

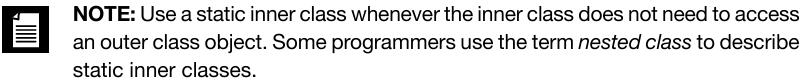
-**static inner class**: exactly like any other inner class, except that an object of a static inner class **doesn’t have** a **reference** to the **outer class object** that generated it.

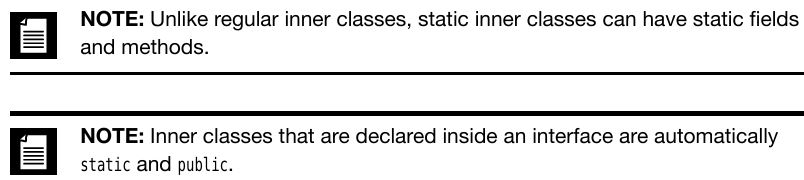
-Use a static inner class when the inner class object is constructed inside a **static method**.





-Note



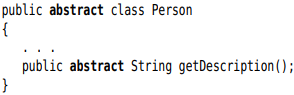


## **2.4 Encapsulation and immutability**

-**Encapsulation**(**information** **hiding**): combine data and behavior in one package and hide the implementation details from users of object. The bits of data in object are **instance fields**, procedures are **methods***.* An object is an instance of class and have values of instance fields: called **state**. The key to making **encapsulation** is to have methods never directly access instance in class other than their own.

## **2.5 Subclass, super class, abstract class**

-Moving up the inheritance hierarch, classes become more general and abstract. There are some attributes that make sense for every class. Using the **abstract**, you don’t need to implement the method:



-**Abstract methods** act as placeholders for method that are implemented in subclasses. It can has **instance fields** and **concrete methods**, or **no abstract method**.

-Abstract cannot be instantiated, you can create object variable but refer a nonabstract class. The variable always refers to an object of subclass.



-Abstract methods are an important concept in Java => inside interfaces.

## **2.6 Method call polymorphically**

-The fact that an object variable can refer to multiple actual types is **polymorphism***.* Automatically selecting the appropriate method at runtime is **dynamic binding**

**-Substituation principle**: you can use a subclass object whenever the program expects a superclass object

-Caution: Arrays of subclass can be converted to arrays of superclass. To make sure no corruption, all arrays remember the element type which they were created

## **2.7 Interface, private, static, default method**

2.7.1 Interface

-All **methods** in **interface** are **automatically** **public**, whether the keyword is specified or not.

-**Interface** is an **abstract class** with **no instance fields**. All fields in Interface are implicitly **public, static and final**, whether the keywords are specified or not.

-Interfaces are not classes. You can never use **new**

****

-But you can **declare**: . An interface variable must refer to implemented object: 

-A **reference** of interface type can be **cast** to any class that implements this interface.

-You can **extend any number of interfaces**.

-An overriding method must return either the **same type** or a **sub type** of the **return type** of overridden method.

-Classes can implement **multiple interfaces**.

+More than one method declaration may be implemented by a singer method declaration if they have the same name.

+If the methods throw exception, the overriding declaration must throw the subclass of the exceptions.

-Extend class and implements interface: only the superclass method matter, the interface is ignored



-**Multiple inheritance of state**: inherit instance fields.

-**Multiple inheritance of type**: implement interfaces and extend classes.

2.7.2 Static and private methods

-Static has a body. You can **redeclare** a static method as a **default** method in sub interface.

-private: You can use private static or instance methods.

2.7.3 Default methods

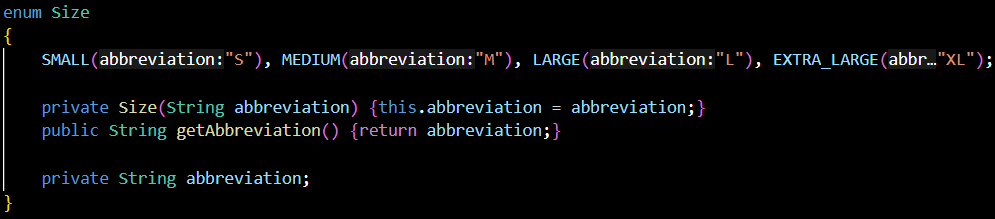
-Default methods supply a **default** implementation with default modifier. It can call other methods.

-It has a body.

-A subinterface can redeclare an inherited default method as **abstract** or provide a different implementation.

## **2.8 Enumeration**





-Enum type is a data type that enables for a variable to be a set of predefined **constants**.-> enum is **final +** Can’t override **clone()**. The enum **values** can be the **same** (MS1(“Ms.”), MS2(“Ms.”))

-enums implicitly extend **java.lang.Enum** -> **cant extend** anything else

-enum defines a class (enum type). The enum body can include methods and other fields. The compiler automatically adds special methods: static **values()** and **valueOf(), ordinal() name().**

-The **constants** are defined 1st, then fields + methods.

-The enum constructor must be **package** or **private** access.

**-Can’t access** **non-final static field** in **constructor**.

-Enums implement **Comparable** in the **natural order** which **they are defined**

# 3. Exception handling (5%)

3.1 try/catch/finally

3.2 Single-catch, multi-catch statements

3.3 Throw, throws

3.4 Try-with-resource

3.5 Custom exception

# 4. Working with Arrays and Collections (10%)

4.1 Array, List, Set, Map, Dequeue

4.2 Comparator, Comparable

4.3 Generic, wildcards

# 5 Working with Streams and Lambda expressions (15%)

5.1 Functional interfaces using lambda expressions

5.2 java.util.function package

5.3 Java Streams filtering, transforming, processing, reduction, grouping, partitioning

5.4 Sequential and parallel streams

# 6. Java Platform Module System (5%)

6.1 Modular vs non-modular application

6.2 Named module, unnamed modules, automatic modules

6.3 Expose, compile, run, deploy module

# 7. Concurrency (4%)

-Deadlock, livelock, starvation recognition

-Runnable, Thread

-Callable, ExecutorService

-java.util.concurrent locking api

# 8. Java I/O API (5%)

8.1 I/O Streams

8.2 NI/O API

8.3 Serialization and deserialization

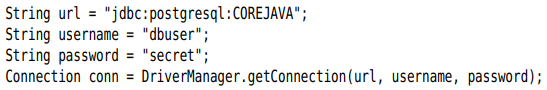
# 9. Securing Coding in Java SE Application (3%)

-Denial of service, code injection, data validation, data integrity

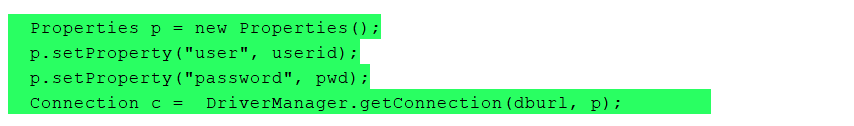
# 10. Database Applications with JDBC (2%)

## **10.1 Database connection, manipulation**

### **10.1.1 Connection**



-Use property:



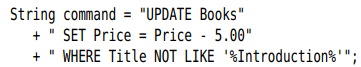
## **10.2 Working with JDBC Statement**

### **10.2.1 Executing SQL Statements**

-Execute a SQL statement, 1st create **Statement** object by **Connection** from **DriverManager.getConnection()**



-Place the statement into a string:



-**executeUpdate()** of Statement interface:



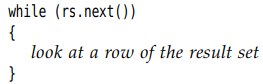
+**executeUpdate()** return a count of rows affected by SQL statement or zero.

+executeUpdate() can execute actions INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE…

-**executeQuery()**: SELECT

+This method returns **ResultSet** object to walk through the result one row at a time:





+The order of rows is arbitrary. You can specify with **ORDER BY**

+Take the contents of the fields:



There are accessors for various type, such as getString() and getDouble(). Each accessor has 2 forms: string (name column) and numeric (number column) argument.

Each get method make type **conversions**. Example: getString() convert any type to string.

-**execute()**: catch-all statement to execute arbitrary SQL statements. It’s commonly used only for queries that a user supplies interactively.

### **10.2.2 Managing Connections, Statements, and Result Sets**

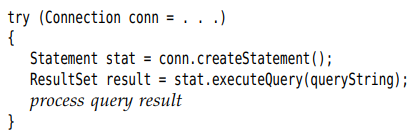
-Every Connection object can create one or more Statement objects. You can use the same Statement object for multiple unrelated commands and queries.

-A statement has at most one open result set. If you want multiple queries concurrently, use multiple Statement.

-There is a limit to the number of statements per connection. Use **getMaxStatements()** of **DatabaseMetaData** interface.

-**close()** when you done using ResultSet, Statement or Connection. **closeOnCompletion()** on Statement close automatically as soon as all its result sets have closed.

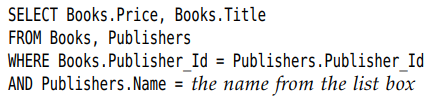
-Use **try-with-resource** with short-lived connection:



## **10.3 Query Execution**

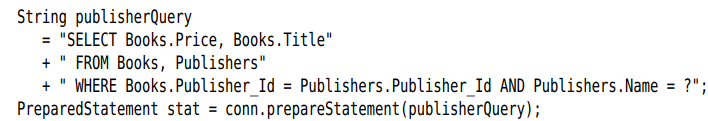
### **10.3.1 Prepared Statements**

-Prepared statements:



+We prepare a query with a host variable and use it many times, each time filling in a different string for the variable. -> improve performance

+Each host variable in a prepared query is indicated with a ?. If there is more than one variable, keep track of the positions of ? when setting the values



-You must bind the host variables to actual values with set(). There are different **set()** for various types

+ 1st argument is the position number of the host variable. 2nd argument is the value.

+If you reuse a prepared query that already executed, all host variables **stay bound** unless you change them with **set()** or **clearParameters()**.

-Once all variables have been bound to values, execute the prepared statement: 

-executeUpdate() return the count of changed rows.



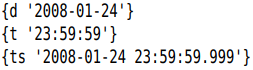
-**Note**:

+**setNull()** must use SQL type in java.sql.Types

### **10.3.3 SQL Escapes**

-“escape” syntax features are supported by databases but use database-specific syntax variations. JDBC driver translates the escape syntax to syntax of database.

-Date and time literals: Use d, t, ts for DATE, TIME or TIMESTAMP values:

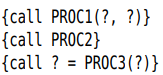


-Scalar function: function returns a single value.

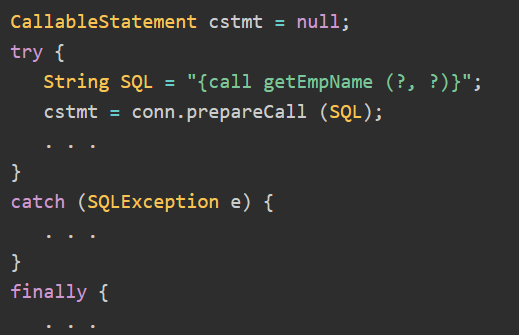
+Embed the standard function name and arguments:



-**Stored procedure**:



+Use **CallableStatement** interface:



Input: set…()

Output:



-Outer join

-\_ and % in LIKE:

+No standard way to use them literally.

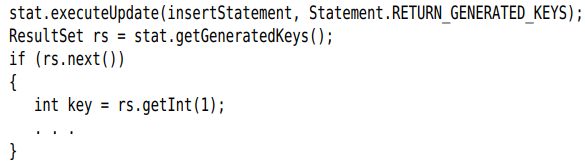
+Example: match all strings containing a \_



### **10.3.5 Retrieving Autogenerated Keys**

-Most databases support mechanisms for autonumbering rows in database. They differ among vendors. The automatic numbers are often used as primary key.

-Retrieving them:



## **10.4 Scrollable and Updatable Results Sets**

-In a **scrollable result**, you can move forward and backward through a result set and jump to any position.

-In an **updatable result set**, you can update entries so that the database is automatically updated.

### **10.4.1 Scrollable Result Sets**

-By default, result sets are not scrollable or updatable.

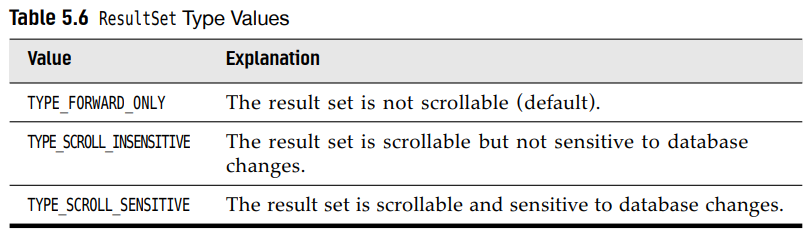
-Must obtain a different Statement object with:

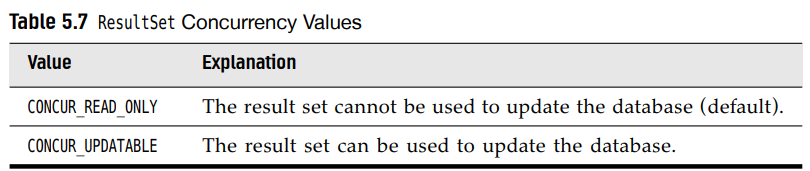


-For a prepared statement:



-**type** and **concurrency**:





-Example: scroll through a result set but don’t edit data:



+Return the result sets: 

are now scrollable. A scrollable result set has a **cursor** that indicates the current position.

+**getType()** and **getConcurrency()** of ResultSet interface to find out what mode a result set actually has.

-**Scrolling**:

+ return true or false

+ move the cursor by any number of rows

+ set the cursor to a row number

+ get the current row number

+first(), last(), beforeFirst(), afterLast()

+isFirst(), isLast(), isBeforeFirst(), isAfterLast()

### **10.4.2 Updatable Result Sets**

-Obtain updatable result sets:



-Note: Not all queries return updatable result sts

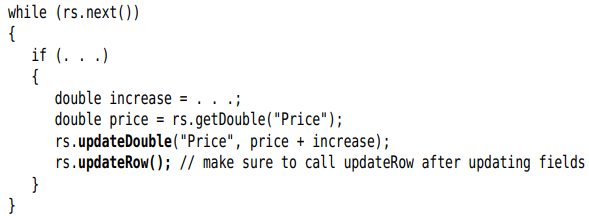
+The query is a join involves multiple tables not updatable

+The query involves only a single table or join multiple tables by their primary key: you should expect the result set to be updatable.

+getConcurrency() to find out.

-**Update**:



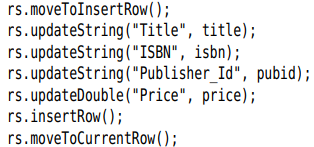


+There are **updateXXX** methods for all data types that correspond to SQL types: updateDouble(), updateString()

+**Note**: If you use updateXXX whose 1st parameter is the column number, be aware that this is the column number in the result set.

+updateXXX changes only the row values, not the database. Use updateRow(). You can also call cancelRowUpdates() cancel the updates to current row

-**Add** a new row to database:



+You can’t influence where the new data is added in the result set or database.

+If you don’t specify a column value in insert row, it’s set to SQL NULL. If the column has a NOT NULL constraint, an exception is thrown and the row is not inserted.

-Delete the row under the cursor: 

-To sum up, java programmers might find it more natural to manipulate the database contents through result sets than by constructing SQL statements.

## **10.5 Transactions**

-You can group a set of statements to form a **transaction**. It can be **committed** when all has gone well-or an error has occurred in one of them, it can be **rolled back** as if none of the statements had been issued.

-The major reason for grouping statements into transactions is **database integrity**.

### **10.5.1 Programming Transactions with JDBC**

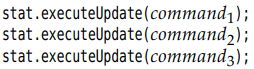
-By default, database connection is in **autocommit mode** –each SQL statement is committed to database as soon as it is executed. Once a statement is committed, you can’t roll it back.

-Turn off the default: 

-Create a statement object in normal way:



-executeUpdate() any number of times:



-If all statements have been executed without error:

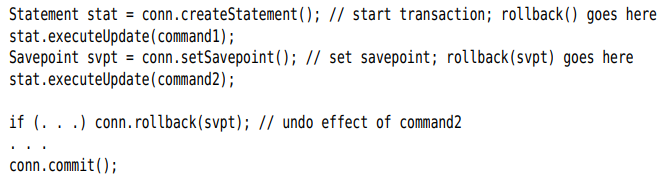


-If an error occurred: 

+All statements since the last commit are automatically reserved. You issue a rollback when the transaction was interrupted by a SQLException

### **10.5.2 Save Points**

-Create a save point marks a point to which you can later return without having to abandon entire transaction.



-If you no longer need a save point: 

### **10.5.3 Batch Updates**

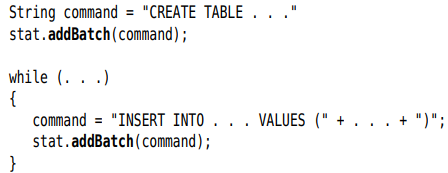
-In a batch updates, a sequence of statements is collected and submitted as a batch.

-**Note**: use **supportsBatchUpdates**() of **DatabaseMetaData** to find out if the databse supports this feature.

-Execute a batch:

+Create a Statement object: 

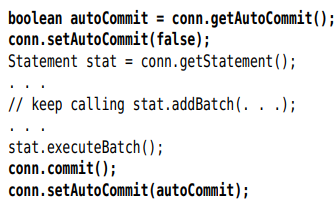
+**addBatch**() instead of executeUpdate():



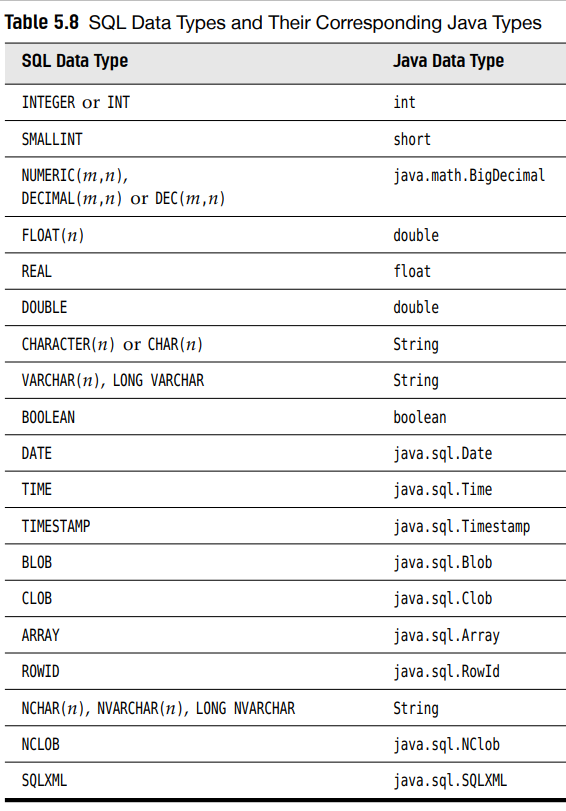
+Submit the entire batch: 

returns an array of row counts for all submitted statements.

-Treat the batch execution as a single transaction. If a batch fails in the middle, roll back to the state before the beginning of the batch.



### **10.5.4 Advanced SQL Types**



# 11. Localization (2%)

11.1 Locale

11.2 Resource bundle

11.3 Message format

11.4 Date format

11.5 Number format

# 12 Annotation (4%)

12.1 Built-in annotation

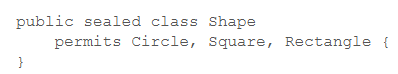
12.2 Annotation creation, applying

# 13.Record and Sealed Class

## **13.1 Sealed class**

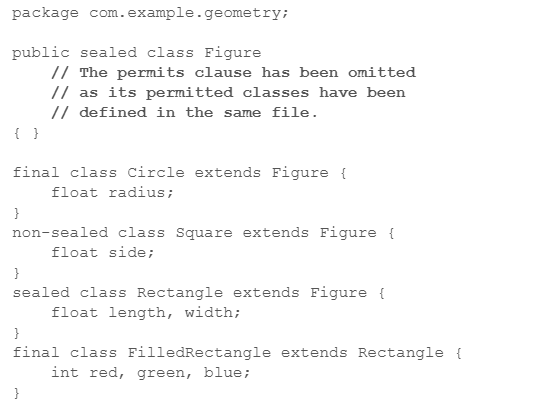
-Sealed classes and interfaces: restrict which other classes or interfaces may extend or implement them.

-Declare:



+The permitted subclasses are in the **same module** or in the **same package.**

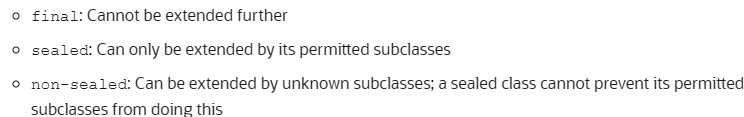
+You can define permitted subclasses in the **same file**, then you **can omit permits**:



-**Permitted subclasses**:

+Must **extend** the **sealed class**.

+Must be one of:



+Must be in the **same module** or **same package**.

-**Sealed interface**: specifies classes and interfaces that can extend and implement

-**Record** classes as **permitted subclasses** of a **sealed interface**: record is **final**

-**java.lang.Class methods**:

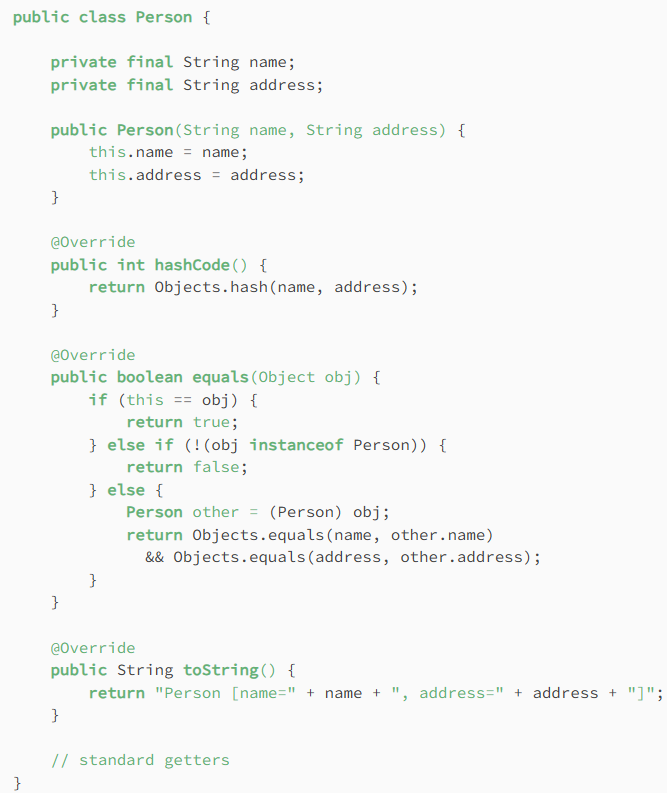
permittedSubClasses()

isSealed()

## **13.2 Record**

13.2.1 Purpose

-We write classes to hold data (database result, query result, information from a service), this data is **immutable**.

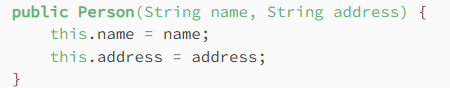


13.2.2 The basics

-JDK 14: Records are immutable data classes that requires only the type and name of fields. final equals(), final hashCode(), final toString(), private, final fields and constructor are generated by Java compiler.



-The equivalent constructor:



+Instantiate objects from record:



-The equivalent **getters** (Note: name of getter = name of instance fields, example: name()), equals (true if objects of same type and values match), hashCode (return same value for 2 objects if all field values match), toString(name of record+field names+field values)

-Record **can’t be extended** and **can’t have extend clause**.

-Record can implement Serializable interface

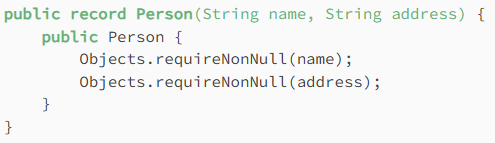
-Record may have at **most 1 varagrs** component.

* + 1. Constructors

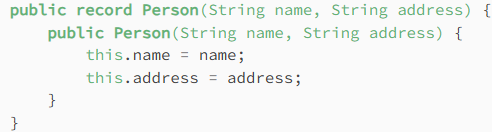
-**Canonical constructor**:

+We can **customize** constructor for **validation**: (**compact form**)

+Example: fields aren’t null

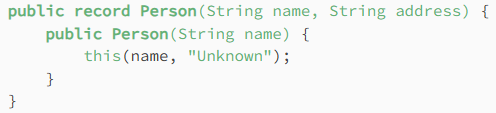


+Creating a constructor with **same arguments** as the generated public constructor is valid, but each field is **manually initialized**: **regular form canonical constructor**



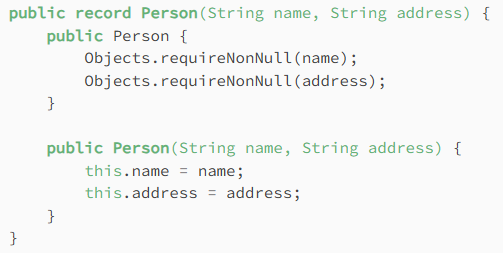
Or public Journal {id=id+1}

-**Non-canonical constructor**: Create **new constructors** with different arguments by supplying either a **canonical constructor** or **another constructor**:



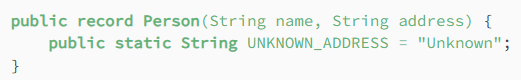
-**Error** when declare **2 canonical constructors**:

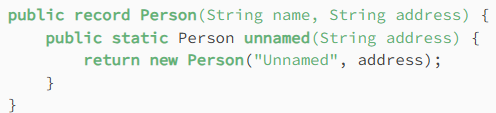
+Example: compact form + regular form



13.2.4 Static variables and methods

-We can use **static variables** and **static (and instance, not setter)** **methods** in record







# 14. Date and Time API

## **14.1 Time Line**

-**Instant** represents a point on the time line.

+An origin (**epoch**) is set at midnight of 1/1/1970 at the prime meridian passing through Greenwich Royal Observatory in London. Time is measured in 86400 seconds per day, to **nanosecond** precision. Instant values go back as far as a 1 billion years.

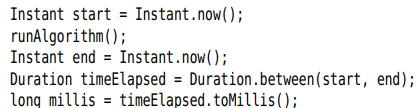
+**Instant.MAX** is 31/12/1000000000

+**Instant.now()** give the current instant.

+**Instant.truncatedTo(ChronoUnit):** make a copy

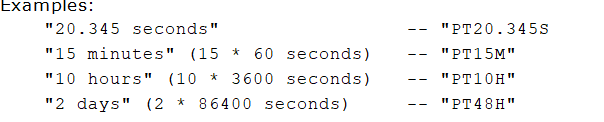
+You can compare 2 instants with **equals()** and **compareTo().**

+**Duration.between()** find out the difference between 2 instants. Example: measure the running time algorithm:

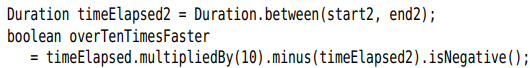


-Duration is the amount of time between 2 instants. You can get the length of Duration by **toNanos(), toMillis(), getSeconds(), toMinutes(), toHours(), toDays()**

**+toString() format**: PT…H…M…S



+Duration API has methods to carry out arithmetic.

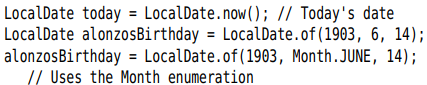




## **14.2 Local Date**

**TemporalAccessor** interface (get)-> **Temporal** interface (plus, minus, until, with) -> LocalDate,Time… + **TemporalAdjuster** interface

-**Construct** LocalDate with **now(**) or static **of()**:



-**Compute** the day of the year:



-**Period** express numbers of elapsed **years, months, days**. **toString() Period** format: P…D

+  or 

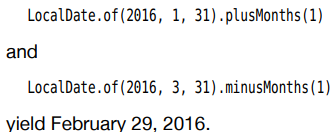
+**until()** yields the different between 2 local dates:



+Find the number of **days**:



-**Caution**: some methods in LocalDate could create non-existen dates:



-**getDayOfWeek()** yields the weekday as a value of **DayOfWeek** enumeration:

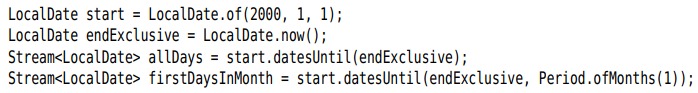


+DayOfWeek enumeration have **plus()** and **minus()**:



+**Note**: java.util.Calendar: Sunday has value 1.

-2 methods **datesUntil**() yield **streams** of LocalDate:



-There are also classes **MonthDay**, **YearMonth**, **Year** to describe **partial dates**.

## **14.3 Date Adjuster**

-**TemporalAdjusters** class provides a number of static methods for common adjustments. Pass the result of an adjustment method to **with(TemporalAdjuster)** or **TemporalAdjuster.adjustInto()**





## **14.4 Local Time**

-create an instance with **now()** or **of()**:



-**plus()** and **minus()** wrap around a **24-hour** day.



-**Note**: LocalTime doesn’t concern itself with AM/PM

-**LocalDateTime** class represent a date and time. This class is suitable for storing points in time in a fixed time zone (schedule of classes or events)

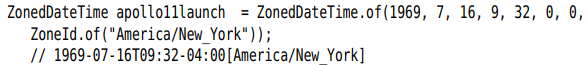
## **14.5 Zoned Time**

-Each time zone has an **ID** (America/New\_York, Europe/Berlin). **ZoneId.getAvailableZoneIds()** to find out all available time zones.

-**ZoneId.of(id)** yields a ZoneId object. You can use it to turn **LocalDateTime into ZonedDateTime** by **local.atZone(zoneId)**

**-**Construct a ZonedDateTime:



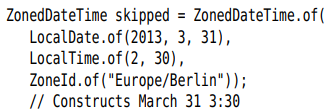


+**apollolaunch.toInstant()** to get the Instant.

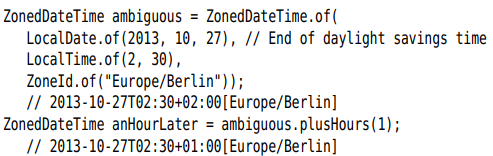
+ get **ZonedDateTime** from **Instant**, use **ZoneId**.

-Many methods of **ZonedDateTime** are the same as those of **LocalDateTime**

-**Daylight savings time**: clocks advance by an hour.

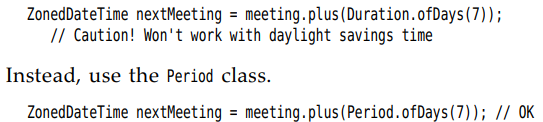


+When daylight time ends, clocks are set back by 1 hour, there are 2 instants with same local time.



The time has the same hours and minutes, but the zone offset has changed.

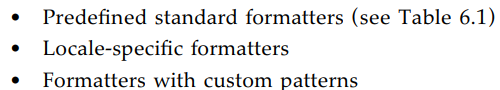
+Pay attention when adjusting a date across daylight savings time boundaries. Example: set a meeting for next week, don’t add 7 days:

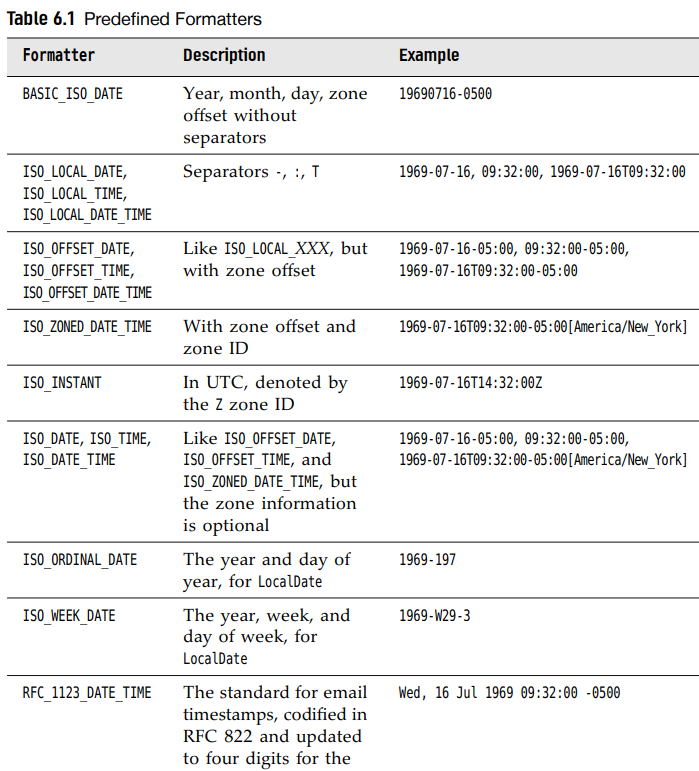
  
+between **ZonedLocalDate** in **DST** and not in DST: need to +1 hour to between time.

-**Caution**: **OffsetDateTime** represents times with an offset from UTCm without time zone rules.

## **14.6 Formatting and Parsing**

-**DateTimeFormatter** class provides 3 kinds of formatters to print date/time value:

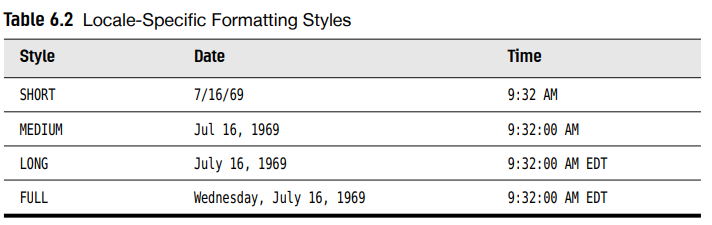




-Use one of the standard formatters:



-To present dates and times to human readers, use locale-specific formatter. There are 4 styles: SHORT, MEDIUM, LONG, FULL



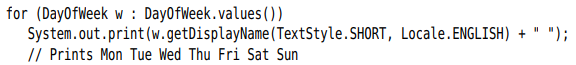
-Static ofLocalizedDate(), ofLocalizedTime(), ofLocalizedDateTime() create a formatter:



+These methods use default locale. Change **withLocale()**

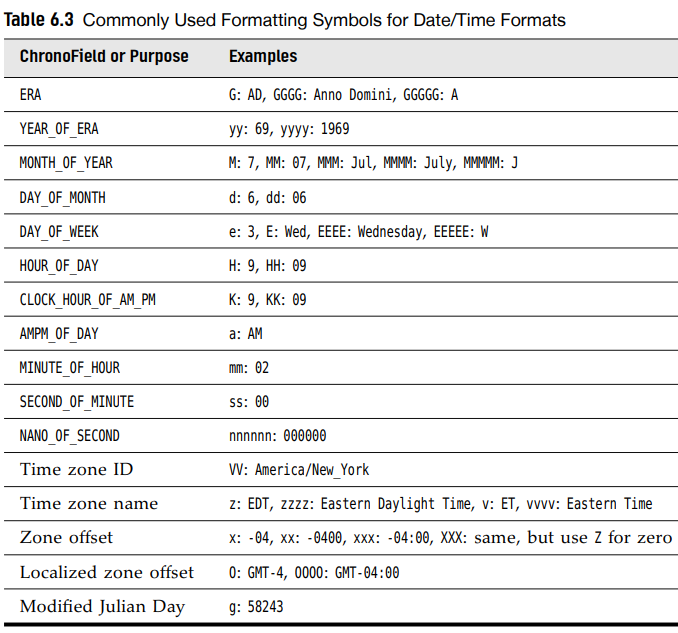


-**DayOfWeek** and **Month** enumerations have **getDisplayName()** for giving info in different locales and formats:

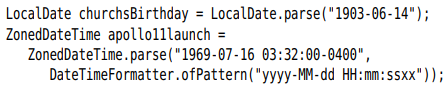


-**Note**: java.time.format.DateTimeFormatter class is intended as a replacement for java.util.DateFormat. Call formatter.toFormat()

-You can roll **your own date format** by specifying a **pattern**: 



-Parse a date/time value from a string: **parse()**



+1st call use standard ISO\_LOCAL\_DATE formatter. 2nd one use a custom formatter.