1. **Concurrency (java.til.concurrent)**
   1. **Time slicing**: processing time for a single execution core is shared among processes and threads.
      1. 1 execution core 🡪 1 thread at any given time.
      2. Most computers have >1 core, but concurrency is still possible.
   2. 2 basic units of execution
      1. **Processes**: programs or running applications.
         1. Have a self-contained execution environment.
            1. **Inter Process Communication (IPC)**: can run multiple processes, possibly on different systems.

Pipes

Sockets

* + - * 1. Have ≥1 threads.

**Main thread**: has the ability to create multiple threads.

* + - 1. Normally have a complete, private set of basic runtime memory.
      2. Example: Java virtual machine (JVM)
    1. **Threads** **(*lightweight processes)***: provide an execution environment.
       1. Multithreaded execution: focus in Java.
       2. Exist within a process
          1. Share…

Memory.

Open files.

Communication of fields and object references

Efficient, but potentially can result in 2 errors

**Thread interference**: when operations, running in different threads acting on the same data ***interleave*** (overlap sequence of steps).

**Counter.java**:even simple statements can translate to multiple steps by the virtual machine.

Unpredictable

Difficult to detect and fix

**Memory consistency errors**: when different threads have inconsistent views of what should be the same data.

Complex causes, but must avoid.

**Happens-before relationship**: guarantee that memory writes by one specific statement are visible to another specific statement.

**Synchronization**:tool employed to prevent thread communication errors.

**Thread contention**: error caused by synchronization when threads try to access the same resource simultaneously

Java runtime executes threads more slowly or even suspend execution.

Memory consistency errors: 2 actions to create a happens-before relationship.

When a statement invokes start(), every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread.

When a thread terminates and causes a join() in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join().

**Intrinsic (monitor) lock**: internal entity involved in enforcing exclusive access to state, as well as establishing happens-before relationships.

Included in every object; automatically obtained from synchronized methods.

Convention: thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them (*owning*), and then *release* the intrinsic lock when it's done with them.

As long as a thread owns an intrinsic lock, no other thread can acquire the same lock.

Others thread will blocked when it attempting to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock, even if return was by uncaught exception.

**Synchronized methods**: strategy for preventing thread interference and memory consistency errors.

Object count instanceof SynchronizedCounter: 2 effects.

Not possible for 2 invocations of synchronized methods on the same object to interleave.

When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Constructors: **CANNOT** be synchronized.

Only the thread that creates an object should have access to it while it is being constructed.

**Reentrant synchronization**: allowing a thread to acquire the same lock more than once.

A thread cannot acquire a lock owned by another thread, but it *can* acquire a lock that it already owns.

Synchronized code invokes a method that also contains synchronized code and both sets of code use the same lock.

Without it, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

* + - * 1. Require fewer resources than processes.
      1. 2 ways of creating a thread: invoke Thread.start().
         1. *Provide a*Runnable *object*: Runnable interface defines a single method, run(), meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the HelloRunnable.

More general, because the Runnable object can subclass a class other than Thread.

* + - * 1. *Subclass Thread*: Thread class itself implements Runnable, though its run() method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread.

Easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.

* + - 1. **Interrupts**: indication to a thread that it should stop what it is doing and do something else.
         1. Interrupted thread must support its own interruption.

If the thread is frequently invoking methods that throw InterruptedException it returns from the run method after it catches that exception

Methods that throw InterruptedException, such as sleep, are designed to cancel current operation and return immediately when an interrupt is received

* + - * 1. Programmers decide how a thread responds to an interrupt; typically used to terminate.
      1. Methods
         1. Static methods: Static methods, which provide information about, or affect the status of, the thread invoking the method.

interrupted(): checks for an interrupt; clears interrupt status.

* + - * 1. Other methods: invoked from other threads involved in managing the thread and Thread object.

sleep(): causes the current thread to suspend execution for a specified period.

Sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS.

2 overloaded versions

specifies the sleep time to the millisecond

specifies the sleep time to the nanosecond

Uses

Efficient means of making processor time available to other threads or processes.

Can also be used for pacing.

Can be terminated by interrupts.

isInterrupted(): used by one thread to query the interrupt status of another, does not change the interrupt status flag.

join(): allows one thread to pause for the completion of another.

Can be overloaded to specify OS-dependent time.

Responds to an interrupt by exiting with InterruptedException.

* + - 1. States
         1. **new**: beginning of thread lifecycle until the program starts thread.
         2. 2 OS-dependent states.

**runnable**: when program starts thread; executing a task.

**ready**: when a thread transitions from new to runnable.

**dispatching the thread**: OS assigns it to a processor.

**Quantum (timeslice)**: small amount of processor time assigned so it can perform its task.

When the quantum expires, the thread is returned to ready state and the OS assigns another thread to the processor.

**Thread scheduling**: transitions between the ready and running states handled by OS.

* + - * 1. **waiting**: when waiting for another thread to perform a task.

Transitions back to runnable when another thread notifies it to continue.

Cannot use processor.

* + - * 1. **time waiting**: enter a timed waiting state for a specific period of time.

Transitions back to runnable when the time interval expires or when the event it’s waiting for occurs

Cannot use processor.

* + - * 1. **blocked**: when attempting to perform a task that cannot be completed immediately and must wait until the task completes.

Cannot use processor.

* + - * 1. **terminated (dead)**: end of state transitions, when thread successfully completes its task or terminates.
      1. Priorities
         1. Thread priority is inherited from the thread that created it.
         2. Thread priorities do not guarantee the order that threads execute.
      2. Scheduling: OS thread scheduler is in charge of which thread runs next.
         1. Highest priority thread is kept running at all times.
         2. If more than one highest priority thread exists, use a round-robin method for a quantum for each thread.
         3. **Preemptive scheduling**: if a higher priority thread enters the ready state, the OS generally preempts the currently running thread.

**Starvation (indefinite postponement)**: higher priority threads continue to influx, postponing a lower-priority thread potentially indefinitely.

**Aging**: OS technique used to increase priority the longer a thread waits.

**Deadlock**: threads that are stopped to wait for each other cannot proceed; hence, neither will ever proceed.

* 1. **Atomic Access**
     1. **Atomic actions**: effectively happen all at once.
        1. Cannot stop in the middle
           1. happens completely
           2. doesn't happen at all
        2. No visible side-effects until the action is complete or fails.
        3. Examples
           1. Reads and writes for reference variables and for most primitive variables (except long and double).
           2. Reads and writes for *all* variables declared volatile (*including* long and double)
        4. Cannot be interleaved, but does not eliminate need to synchronize.
           1. No fear of thread interference.
           2. Can still have memory inconsistency errors.

Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable.

Changes to a volatile variable and the side effects of the code that led up the change are always visible to other threads.

* + - 1. Using simple atomic variable access…
         1. is more efficient than accessing variables through synchronized code.
         2. requires more care to avoid memory consistency errors.
         3. is worthwhile depending on the size and complexity of the application.

1. **XML Parsing**: going through an XML document to access data or to modify data in one or other way.
   1. **Extensible Markup Language (XML)**: simple text-based language recommended by **World Wide Web Consortium (W3C)** to store and transport data in plain text format.
      1. Properties
         1. markup
         2. tag-based
      2. Advantages
         1. **Technology agnostic**: being plain text, XML is technology-independent.
         2. **Human-readable**: uses simple text format, and therefore is human-readable and understandable.
         3. **Extensible**: no predefined tags; customize self-descriptive tags.
         4. **Allows validation**: using XML Schema Definition (XSD), document type definition (DTD) and XML structure can be validated easily.
      3. Disadvantages
         1. **Redundant Syntax**: contains lot of repetitive terms.
         2. **Verbose**:file size increases the transmission and storage costs.
   2. **XML Parser**: provides methodology to access/modify XML data.
      1. Java: multiple options
         1. **Document Object Model (DOM) Parser**: parses the document by loading the complete contents of the document and creating its complete hierarchical tree in memory.
            1. **Document Object Model (DOM)**: defines an interface that enables programs to access and update the style, structure, and contents of XML documents.

Recommendation of W3C

Interfaces

**Node**: base datatype.

**Element**: majority of the objects interacting with in the source code.

**Attr**: attribute of an Element.

**Text**: actual content of an Element or Attr.

**Document**: entire XML document.

Methods to examine the contents and structure of the Document

**Document.getDocumentElement()**: returns the root Element of the Document

**Node.getFirstChild()**: returns the first child of a given Node.

**Node.getLastChild()**: returns the last child of a given Node.

**Node.getNextSibling()**: returns the next sibling of a given Node.

**Node.getPreviousSibling()**: returns the previous sibling of a given Node.

**Node.getAttribute(attrName)**: for a given Node, returns the Attr with the requested name.

* + - * 1. Characteristics

Need to know a lot about the structure of a document.

Need to move parts of the document around.

Need to use the information in the document more than once.

* + - * 1. Using DOM Parsing

**Import XML-related packages**

import org.w3c.dom.\*;

import javax.xml.parsers.\*;

import java.io.\*;

**Create a DocumentBuilder**

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

DocumentBuilder builder = factory.newDocumentBuilder();

**Create a Document from a File or Stream**

StringBuilder xmlStringBuilder = new StringBuilder();

xmlStringBuilder.append("<?xml version="1.0"?> <class> </class>");

ByteArrayInputStream input = new ByteArrayInputStream( xmlStringBuilder.toString().getBytes("UTF-8"));

Document doc = builder.parse(input);

**Extract the root Element**

Element root = document.getDocumentElement();

Examine attributes

//returns specific attribute

getAttribute("attributeName");

//returns a Map (table) of names/values

getAttributes();

**Examine sub-Elements**

//returns a list of subelements of specified name

getElementsByTagName("subelementName");

//returns a list of all child nodes

getChildNodes();

* + - 1. **Simple API for XML (SAX) Parser**: parses the document on event-based triggers but does not load the complete document into the memory.
      2. **Java-based Document Object Model (JDOM)** **Parser**: parses the document in similar fashion to DOM parser but in an easier way.
      3. **Streaming API for XML (StAX) Parser**: Parses the document in similar fashion to SAX parser but in a more efficient way.
      4. **XML Path Language (XPath) Parser**: parses the document based on expression and is used extensively in conjunction with XSLT.
      5. **DOM4J Parser**: Java library to parse XML, XPath and XSLT using Java Collections Framework.
         1. Provides support for DOM, SAX and JAXP.

1. **Databases**: store data so that information can be retrieved from it
   1. **Database Management System (DBMS)**: handles the way data is stored, maintained, and retrieved in a database.
   2. **Relational databases**: present information in tables with rows and columns.
      1. **Relational Database Management System (RDBMS)**: handles the way data is stored, maintained, and retrieved in a relational database.
         1. Microsoft SQL Server
         2. Oracle
         3. Sybase
         4. IBM DB2
         5. Informix
         6. PostgreSQL
         7. MySQL
         8. Java DB, Oracle’s version of Apache Derby
      2. Relation: collection of objects of the same type (rows).
         1. A table as a relation.
            1. Data can be related according to common keys or concepts
            2. Basis for relational databases: ability to retrieve related data.
      3. Characteristics
         1. **Integrity rules**: ensure the data in the database are accurate and always accessible.
            1. Rows in relational tables should all be distinct.

duplicate rows creates issues with resolving which of two possible selections is the correct one.

Most DBMSs allow for specifying that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

* + - * 1. Column values must not be repeating groups or arrays.

Any column that is part of a primary key cannot be null.

* + - * 1. null values: used to indicate that a value is missing.

Does **not** equate to a blank or zero.

Two null values are **not** considered equal.

Blank considered equal to another blank

0 equal to another 0.

* + - * 1. Entity integrity via primary key

using one or more columns to identify a particular row

any column that is part of a primary key cannot be null

* 1. **Java Database Connectivity (JDBC) API**
     1. Used to make database tasks easy.
        1. Execute common SQL statements
        2. Perform other objectives common to database applications
        3. Access any kind of tabular data.
        4. Access data stored in a relational database.
     2. Used to develop Java applications to manage programming activities.
        1. Connect to a data source, like a database.
        2. Send queries and update statements to the database.
        3. Retrieve and process the results received from the database in response to a query.
     3. Process
        1. Instantiates a DriverManager to connect to a database driver and log into the database.
        2. Instantiates a Statement that carries the SQL language query to the database.
        3. Instantiates a ResultSet that retrieves the results of the query.
        4. Executes a while loop to retrieve and display the results.
     4. Sample code

public void connectToAndQueryDatabase(String username, String password) {

Connection con = DriverManager.getConnection( "jdbc:myDriver:myDatabase", username, password);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

while (rs.next()) {

int x = rs.getInt("a");

String s = rs.getString("b");

float f = rs.getFloat("c");

}

}

* + 1. Architecture
       1. **2-tier model**



* + - * 1. Java applet or application talks directly to the data source.
        2. Requires a JDBC driver that can communicate with the particular data source being accessed.
        3. SQL commands are delivered to the database or other data source.
        4. Results of those statements are sent back to the application
        5. Data source may be located on another machine to which the application is connected via a network; known as a **client/server configuration**, the user's machine as the client, and the machine housing the data source as the server.
      1. **3-tier model**



* + - * 1. SQL commands are sent to a "middle tier" of services; middle tier sends the commands to the data source.
        2. Data source processes the commands and sends the results back to the middle tier, which then sends them to the user.
        3. Advantages

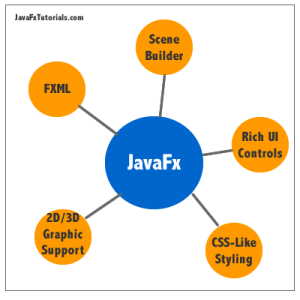
Possible to maintain control over access and updates that can be made to data.

Simplifies the deployment of applications.

Performance advantages

* + 1. Usage
       1. Download and include the appropriate JDBC driver library.
          1. <*someDatabase*>.jar
       2. Create a class specific to connecting to a database
          1. PostgreSQL
          2. MySql
          3. SqlServer
       3. Create class(es) to interface with the database.
       4. Develop a UI that allows users to interact with the data in the database.

1. **User Interface (UI)**
   1. **Abstract window toolkit (awt**): made a program with a basic UI.
      1. Uses
         1. **Applets**: embedded Java programs on webpages.
         2. Applications
      2. Elements
         1. Buttons
         2. Checkboxes
         3. Text fields
   2. **Swing**: Java’s updated **graphical UI (GUI)** library with more tools and an updated look and feel.
      1. Popular
   3. **JavaFX**: powerful way to control the design of your application with CSS-like syntax and precision.
      1. Oracle’s replacement for Swing.
         1. 2005: Sun acquired the technology when they bought out.
         2. 2007: first release of JavaFx, didn’t catch on.
         3. 2009: Oracle acquired Java from Sun Microsystems.
         4. 2010: Oracle announced the future direction for Java which included a completely redone JaxaFx as the centerpiece technology for building applications.
         5. 2014: Java 8 released in and it is the first MAJOR Java release that includes JavaFx as an integral part of its core distribution.



* + 1. Features
       1. New and improved interface tools.
          1. Bar and pie charts
          2. Cool things you can find using JQuery like date-pickers, accordion panes, tabbed panes, etc.
          3. Media player and web-rendering controls.
       2. **FXML**: new HTLM-like language used only to define the interface of an application, keeping it completely separate from the code logic.
          1. Same concept as C# and XAML by Microsoft.
       3. **Scene Builder**: outstanding application which can be integrated into NetBeans and Eclipse to create and generate FXML documents using a drag-and-drop design interface
          1. Similar to Dreamweaver for HTML.
       4. Integrated library for graphics (both 2D and 3D) as well as animation tools that rival Flash, JQuery and CSS animations.
          1. Responsive UI
          2. Animation/visual effects
          3. Gradient color effects like in CSS
       5. Mobile platform development tools
          1. **JavaFX ports**: open-source project that helps bridge the path to mobile application development with JavaFx.
       6. Open-source
          1. Survival chances greatly improved with a worldwide army of JavaFx developers contributing to the development and well-being.
          2. Others are quickly developing additional tools, plugins, and UI controls that go beyond the offerings at Oracle.
    2. In NetBeans
       1. **File**🡪**New Project**
          1. **JavaFX** application category.
          2. Choose **JavaFX FXML Application**.
          3. Click **Next**.
       2. Name the project, and click **Finish**.
       3. NetBeans IDE opens an FXML project that includes the code for a basic; includes 3 files:
          1. **FXMLExample.java:**  takes care of the standard Java code required for an FXML application
          2. **Sample.fxml:**  FXML source file in which you define the user interface.
          3. **SampleController.java:**  controller file for handling the mouse and keyboard input.
       4. Enter **fxml\_example** and click **OK**.
    3. Architecture
       1. **JavaFX platform**: engine that runs the JavaFX code.
          1. **Prism**: a JavaFX high performance graphics engine.
          2. **Glass**: a small and efficient windowing system.
          3. **Media engine**
          4. **Web engine**
       2. **Scene Graph**: hierarchical tree of nodes that represents all of the visual elements of the application's UI.
          1. Starting point for constructing a JavaFX application.
          2. Can handle input and can be rendered.
          3. **Node**: single element in graph.

Has…

ID

style class

bounding volume

(with the exception of root node), a single parent and ≥0 children.

Can have…

Effects

Blurs

Shadows

Opacity

Transforms

Event handlers

Mouse

Key

Input method

Application-specific states

* + - * 1. Includes…

Graphics primitives

Rectangles

Text

Controls

Layout containers

Images and media

* + - * 1. Simplifies working with rich UIs.
      1. **Java Public APIs for JavaFX Features**
         1. APIs

**javafx.animation APIs**: can animate various graphics.

**javafx.scene API**: allows creation and specification of content.

Nodes: shapes (2-D and 3-D), images, media, embedded web browser, text, UI controls, charts, groups, and containers.

State: transforms (positioning and orientation of nodes), visual effects, and other visual state of the content.

Effects: simple objects that change the appearance of scene graph nodes, such as blurs, shadows, and color adjustment.

* + - * 1. Features

Use of powerful Java features, such as generics, annotations, multithreading, and Lambda Expressions.

Make it easier for Web developers to use JavaFX from other JVM-based dynamic languages, such as Groovy and JavaScript.

Allow Java developers to use other system languages, such as Groovy, for writing large or complex JavaFX applications.

Allow the use of binding which includes support for the high performance lazy binding, binding expressions, bound sequence expressions, and partial bind reevaluation. Alternative languages (like Groovy) can use this binding library to introduce binding syntax similar to that of JavaFX Script.

Extend the Java collections library to include observable lists and maps, which allow applications to wire user interfaces to data models, observe changes in those data models, and update the corresponding UI control accordingly.

* + - 1. **Graphics System**
         1. Supports both 2-D and 3-D scene graphs
         2. 2 graphics accelerated pipelines

**Prism**: processes render jobs.

Can run on both hardware and software renderers, including 3-D.

Responsible for rasterization and rendering of JavaFX scenes

**Quantum toolkit**: ties Prism and Glass Windowing Toolkit together and makes them available to the JavaFX layer above them in the stack

Manages the threading rules related to rendering versus events handling.

* + - 1. **Glass Windowing Toolkit**: platform-dependent layer that connects the JavaFX platform to the native OS.
         1. Main responsibility: provide native OS, such as managing the windows, timers, and surfaces.
         2. Responsible for managing events queue.
         3. Responsible for executing the pulse events using high-resolution native timers to make the execution
      2. **JavaFX Threads**: multiple threads run by the system at any given time.
         1. **JavaFX application thread**: primary thread used by JavaFX application developers.

Any **live scene** (part of a window) must be accessed from this thread.

Enables developers to create complex scene graphs on a background thread while keeping animations on live scenes smooth and fast.

* + - * 1. **Prism render thread**: handles the rendering separately from the event dispatcher.

Allows frame *N* to be rendered while frame *N* +1 is being processed.

Concurrent processing:a big advantage, especially on modern systems that have multiple processors.

* + - * 1. **Media thread**: runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.
      1. **JavaFX Pulse**: event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements on the scene graph with Prism.
         1. Throttled at maximum 60 frames per second (60 FPS).
         2. Fired whenever animations are running on a scene.

Even when animation is not running, a pulse is scheduled when something in the scene graph is changed.

State of the elements on the scene graph is synchronized down to the rendering layer.

* + - * 1. Layout and CSS: also tied to pulse events; the system automatically performs a CSS and layout pass once per pulse to avoid performance degradation.
      1. **JavaFx Media and Images (javafx.scene.media APIs)**: support for visual and audio media.
         1. File types

Audio

.MP3

.AIFF

.WAV

Video

.FLV

* + - * 1. Functionality: 3 components

Media: media file.

MediaPlayer: plays media file.

MediaView: node that displays media.

* + - 1. **Web Engine Component**
         1. **WebKit**: open-source web browser engine that supports…

HTML5

CSS

JavaScript

DOM

SVG

* + - * 1. Enables developers to implement features

Render HTML content from local or remote URL

Support history and provide Back and Forward navigation

Reload the content

Apply effects to the web component

Edit the HTML content

Execute JavaScript commands

Handle events

* + - 1. **Java Cascading Style Sheets (CSS)**: provides the ability to apply customized styling to the UI of a JavaFX application without changing any of that application's source code.
         1. Can be applied to any node in the JavaFX scene graph and are applied to the nodes asynchronously.
         2. Styles: can also be easily assigned to the scene at runtime, allowing an application's appearance to change dynamically.
         3. Based on the W3C CSS version 2.1 specifications, with some additions from current work on version 3
      2. **UI Controls**
      3. **Layout**
      4. **2-D and 3-D Transformations**
      5. **Visual Effects**