Long Exam 3 and 4 Quiz Reviewer

Data types

- define what kind of value a column can hold
  - integer data / character data / monetary data / datetime data

Data definition language

- `CREATE DATABASE [name_of_database];`
  - `SHOW DATABASES;` - show the current database
  - `USE [name_of_database];` - to select the database created
  - `DROP DATABASE database_name;` - delete the database
- `CREATE TABLE` - create a new table in the user's schema
  - `SHOW TABLES;` - show the tables in the current database

```
        CREATE TABLE table_name(
                column_name datatype(size) constraint_name,
        );
```

  - `ALTER TABLE` - modify a table definition
    - add / modify / delete attributes or constraints
    - How to add a column in a table
      - `ALTER TABLE table_name ADD column_name datatype;`
    - How to remove a column in a table
      - `ALTER TABLE table_name DROP COLUMN column_name;`
    - Change the data type of a column in a table
      - `ALTER TABLE table_name MODIFY COLUMN column_name datatype;`
    - How to add a primary key to a table
      - `ALTER TABLE table_name ADD PRIMARY KEY(column_name);`
    - How to add a foreign key to a table
      - `ALTER TABLE table_name ADD FOREIGN KEY(column_name) REFERENCES table_name2;`
  - `CREATE TABLE AS` creates a new table based on a query in the user's database
    - create a new table from existing information
    - `CREATE TABLE new_table AS (SELECT * FROM old_table);`
  - `DROP TABLE table_name;` - permanently delete a table
- `CREATE INDEX` - create an index for a table
- `DROP INDEX` - permanently delete an index

Relations

- Stored relations -> tables
  - also called base relations and base table
- Temporary results -> result sets
- Virtual relations -> **views**
  - a view is a virtual relation on the result set of a select statement
  - contains rows and columns, just like a real table
  - can contain fields from one or more real tables in the database
  - `CREATE VIEW` - create a dynamic subset of rows and columns from one or more tables
    - We can also distinguish between attributes by giving them a different name

```
CREATE VIEW view_name AS SELECT column_list FROM table_name WHERE condition_list;
CREATE VIEW view_name (column1, column2) AS SELECT col1, col2 FROM table_name WHERE condition_list;
```

- **Querying views**
  - A view can be used inside another query or inside another view to present exactly the data that we want to the user
  - `SELECT model FROM blue_cars WHERE year=1989;` is the same as `SELECT model FROM cars WHERE color='blue' AND year=1989;`
- **Modifying views**
  - modifying the underlying tables that make up the view
  - Inserting into a view
    - is possible but would leave the columns not present in the view as NULL
    - `INSERT INTO blue_cars (model, year) VALUES ('my_car', 1990);` would create a new row but the `color` attribute would be NULL
      - To solve, create the view also containing `color` in the column list, and when inserting, also include `blue` as the color
      - `INSERT INTO blue_cars (color, model, year) VALUES ('blue', 'my_car', 1990);`
  - Deleting and Updating from a view - works intuitively, any row matched with the `WHERE` clause is updated / deleted in the underlying table
- `DROP VIEW view_name;` - permanently delete a view
  - does not affect any rows of the underlying relation / table

Constraints

- Used to specify rules for the data in a table
- if there is a violation between the constraint and data action, the action is aborted
- can be specified when the table is created, or using the ALTER TABLE statement
- `NOT NULL` - ensure that a column will not have null values
- `UNIQUE` - ensure that a column will not have duplicate values
  - a UNIQUE column can still be null, although only 1 row can be null
- `PRIMARY KEY` - define a primary key for the table
  - A combination of **NOT NULL** and **UNIQUE**
  - A column / combination of two columns have a unique identity which helps to find a record in the table more easily
- `FOREIGN KEY` - define a foreign key for the table
  - Ensures the referential integrity of the data in one table to match values in another table
- `DEFAULT` - define a default value when none is given
  - Specifies a default value for the column
- `CHECK` - validate the data in an attribute
  - Ensure the value in a column meets a specific condition

Data manipulation language

- Commands
  - `INSERT` - insert rows into a table

    ```
    INSERT INTO table_name (col_name, col_name2) VALUES ('value1', 'value2');
    ```

  - `INSERT SELECT` - insert rows from one table to another table that already exists
    - the table must already exist, and will error if it doesn't
    - **SQL allows you to copy the contents of selected table columns so that the data doesn't need to be re-entered manually, but column characteristics must match**

```
    INSERT INTO target_table (target_columns) SELECT source_columns FROM source_table;
    ```

  - `SELECT` - select attributes from rows in one or more tables or views
    - `GROUP BY` - group selected rows based on on one or more attributes
      - groups rows into smaller collections, the aggregate function will then summarize the data within each smaller collection
      - `HAVING` - restricts the selection of grouped rows based on a condition
        - basically a `WHERE` for a `GROUP BY`
        - `SELECT Email FROM Person GROUP BY Email HAVING COUNT(Email) > 1;`
          - List emails that are duplicated in the Person table
    - `ORDER BY` - orders the selected rows based on one or more attributes

    ```
    SELECT list, of, columns FROM table_name WHERE condition AND condition2 ORDER BY column1 ASC;
    ```

  - `SELECT INTO` - copy the contents of the selected table into a new table
    - **creates a new table if it doesn't exist already**
    - Copy all rows and columns from `old_table` to `new_table`
      - `SELECT * INTO new_table FROM old_table;`
    - Copy all rows and columns from `old_table` to `new_table` in another database
      - `SELECT * INTO new_table IN 'another_database' FROM old_table;`
    - Copy only a few columns into the new table
      - `SELECT columnlist INTO new_table FROM old_table;`
    - Copy only a group of rows into the new table
      - `SELECT * INTO new_table FROM old_table WHERE condition;`
    - Copy data from more than one table into the new table

    ```
    SELECT c.name, o.id INTO CustomerOrders FROM customers AS c LEFT JOIN orders o ON
    c.customer_id=o.customer_id;
    ```

  - `WHERE` - restricts the selection of rows based on a condition
  - `UPDATE` - modify an attribute's values in one or more table's rows

    ```
        UPDATE table_name SET columnname=new_value WHERE condition AND condition;
    ```

  - `DELETE` - delete one or more rows from a table
    - `DELETE FROM table_name WHERE condition_list;`
- Operators
  - `=, <, >, <=, >=, <>, !=` - used in conditional expressions
  - `AND, OR, NOT` - used to join multiple conditional expressions together
  - `BETWEEN` - check whether an attribute is within a range
  - `IS NULL` - check whether an attribute is NULL
  - `LIKE` - check whether an attribute matches a given string pattern
    - *is basically regex* where the `*` matches multiple characters and `_` matches a single character
    - `J*` - matches John and Jabol
  - `IN` - checks whether an attribute value matches any value within a value list
    - `SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'UK');`
  - `ALL` - compares an attribute against all values in a list
    - `SELECT * FROM Teachers WHERE age > ALL (SELECT age FROM Students);`
      - get all teachers whose age is greater than the age of all students
  - `ANY` - like ALL but returns true for a match with just once student
  - `EXISTS` - check whether a subquery returns any rows

    - `DISTINCT` - Limit values to unique values
        - `SELECT DISTINCT column_name FROM table_name;`
    - `TOP` - limit the number of values selected
        - `SELECT TOP 3 * FROM Customers;`
        - `SELECT * FROM Customers LIMIT 3;`
- Aggregations
    - `COUNT` - return the number of rows with non-null values for a given column
        - `SELECT COUNT(column_name) AS cnt FROM table_name WHERE condition;`
    - `MIN` - returns the minimum attribute value found in a given column
    - `MAX` - returns the maximum attribute value found in a given column
    - `SUM` - returns the sum of all values for a given column
    - `AVG` - returns the average of all values for a given col

SQL Aliases

- makes the output more readable
- An alias is an alternative name given to a column or table in any SQL statement
    - can temporarily rename a table or a column name to make them more readable
        - `SELECT column_name AS c FROM table_name;`
        - `SELECT column_name FROM table_name AS t;`
    - Useful when dealing with dealing with multiple tables in a single query as it can make them shorter
    - Used with aggregation functions since by default they take the name of the column
- An alias is sometimes necessary when JOINing two tables, ie, if they have the same column names
- Example
    - `SELECT name, CONCAT(city, postal_code, country) AS address FROM customers;`
        - The column name would be `CONCAT(city, postal_code, country)` without the alias

SQL Join

- Used to combine rows from two or more tables based on a common field between them
- `INNER JOIN` - returns all rows when there is atleast one match in both tables
    - Cannot contain null
- `LEFT JOIN` - returns all rows from the left table and the matched rows from the right table
    - Can contain null in the right side for rows in the left table that do not match with a row on the right side

```sql
-- List the firstName, lastName, city and state of each person
-- and show null if they don't have an address in the Address table
SELECT firstName, lastName, city, state FROM Person LEFT JOIN Address ON Person.personId = Address.personId;


-- List all customers who haven't made orders
-- We use LEFT JOIN to list all customers with any orders, then filter by those who haven't made any.
SELECT c.name AS Customers FROM Customers c LEFT JOIN Orders o ON c.id = o.customerId WHERE o.id IS NULL;
```

- `RIGHT JOIN` - return all rows from the right table and the matched rows from the left table
- `FULL JOIN / FULL OUTER JOIN` - returns all rows when there is a match in one of the tables
    - Can contain null in both sides
    - DOES NOT exist in MySQL
- `UNION` operator
    - used to combine the result-set of two or more select statements
    - the `SELECT`s used in the UNION must have the same number of columns, and similar datatypes, they must also have the same column order

        - the column names of the result set of a UNION is equal the column names in the first select statement in the UNION
    - selects only distinct values by default. To allow duplicates, use `UNION ALL`.

```
SELECT City, Country FROM Customers WHERE name LIKE 'L%'
UNION ALL
SELECT City, Country FROM Suppliers WHERE name LIKE 'J%';
```

SQL Subqueries

- An inner query is a query inside another SQL query
- used to return data that will be used in the main query as a condition to further restrict the data to be retrieved
- can be used with the SELECT, INSERT, UPDATE, and DELETE statements, and with operators
    - usually added within the WHERE clause of another SQL select statement
    - is executed before its parent query
- Guidelines
    - A subquery **must be enclosed in parentheses** and can be named by adding an identifier after the parenthesis
        - CANNOT be implemented as soon as a SELECT keyword is called
    - A subquery must be placed on the right side of their comparison operator
    - They cannot manipulate their results
    - Use single row operators with single row subqueries
- Example:

```
-- List all employees that earn more than their manager
SELECT name AS Employee FROM employee e WHERE e.managerId IS NOT NULL AND
        -- get the salary of their manager in the subquery
    (SELECT salary FROM employee e_1 WHERE e_1.id = e.managerId) < e.salary;
```

ICS2608 Notes

Chapter 1

Web client

- requests a resource (HTML, PDF)
- knows how to communicate and interpret the request and response types of the server

Webserver

- gets the resource requested by the client and returns it in the server
- can be a physical machine or server application
- can only return static content, finds the file requested as is and hands it back to the client
    - Cannot do computations
    - Need a helper application to generate pages just in time / dynamically and save data on the server
        - the page is based on data submitted by the user
        - the page uses information from databases or other server-side sources

Hyper Text Markup Language (HTML)

- is returned by servers
- tell browsers how to present content to user

Hyper Text Transfer Protocol (HTTP)

- protocol used by web server and a browser
- HTTP request it sent by the browser and the server sends the HTTP response back
- **Request** - requests and sends form data to the server
    - HTTP method (the action to be performed)
        - GET method
            - has limited length (dependent on the OS)
            - form data is appended to the URL
                - can be bookmarked (good for search)
                - may not be secure since the form data can be seen in the URL
            - `method="GET"` in HTML form element attributes
            - An anchor tag always sends a GET request
        - POST method
            - has unlimited length
            - form data is in the request body
                - user can't bookmark the resulting URL
                - considered more secure
            - `method="POST"` in the html form element attribute
        - There are other methods
            - PUT and DELETE - used in RESTful applications
            - HEAD, TRACE, OPTIONS, CONNECT - used in CORS
    - Page to access (URL)
        - **Universal Resource Locator (URL)**
            - Has the protocol, domain, port and path
            - Protocol - which protocol to use
            - Server / domain - physical name of the server you're looking for
            - Port - identifies ethe server application. 80 is the default since 80 is the port used by http
            - Resource - The name of the content being requested
    - Form parameters
- **Response**
    - Status code
    - Content-type
    - The content

JavaServer Pages (JSP)

- just like an html page but you can put java code inside of it

```
<%= new java.util.Date() %>
```

Chapter 2

WebContainer

- servlets don't have a main method since they are under the control of the WebContainer, also called the Application server (glassfish, tomcat)
- **what do they do?**
    - Communication support
        - WebContainers know HTTP and how to communicate with the webserver
    - Lifecycle management - life and death of servlets
    - Multithreading - creating threads for every request
    - Declarative security - no need to recompile your code since configuration is done with an XML file
    - JSP support

- **How Requests are handled**
  - it creates an `HttpServletRequest` and `HttpServletResponse` objects, and finds the correct servlet based on the URL in the request
  - the `service()` method of the servlet is called, which picks which method handler to call depending on the method of the request
    - `doGet()` or `doPost()` is run, which stuffs the page into the response object, so the response goes back to the container
  - The container converts the response object into an HTTP response and sends it back to the client
    - request and response objects are GC'd
- **Classes and interfaces**
  - `Servlet` interface - contains mostly lifecycle methods
    - `init()` - executed once when the servlet is first loaded and before the servlet can service any client requests
      - a unique ServletConfig is created for the servlet
      - gives you a chance to init your servlet before handling any client requests
      - container reads the servlet init parameters from the Deployment Descriptor and gives them to servlet config, which is passed into the init method
    - `service()` - looks at the request and figures out what it should run
      - when a request is sent, the container starts a thread and invokes service().
      - service() looks at the request and invokes the matching `do<Method>()` post in the servlet
    - `destroy()` - called when the server deletes the servlet instance, not called after every request
      - cleaning resources
    - `getServletConfig()`
    - `getServletInfo()`
  - `GenericServlet` - an abstract class that implements a lot of the work that is common between different types of servlets
    - most of the secret behavior comes from this class
  - `HttpServlet` - extends GenericServlet and adds HTTP specific methods
    - implements the service method to reflect HTTP specific behavior
  - `MyServlet` - the place where you override the HTTP methods that you used
    - `doGet()` and `doPost()` - where you should actually put your application code, usually have a common method that is called by both these methods
- **Web.xml** examples
  - Web.xml is also called **Deployment Descriptor** or DD
  - **ServletConfig**
    - Allows us to not hardcode values by placing them in a configuration file that can be read by servlets
    - If these data changes, no need to modify and recompile the servlet
    - The config of a servlet is specific to that servlet
      - Creating a new init param
        - Using the parameter: `getServletConfig().getInitParameter("email")`

```
    // this is located inside a servlet tag, it's only available for that servlet
    <init-param>
            <param-name>Insert the key here</param-name>
            <param-value>Insert the value here</param-value>
    </init-param>
```

  - **ServletContext** - exposes a parameter to all servlets in the same web application
    - Creating a new context param
      - parameters that are available to the entire application
      - `getServletContext().getInitParameter("the_key")`

```
    <context-param>
            <param-name>Insert the key here</param-name>
```

```
        <param-value>Insert the value here</param-value>
    </context-param>
```

- **Servlet mappings**
    - Allows you to change the organization of your packages and servlets without having to change URLs in HTML and JSP
    - Also prevents clients from knowing the directory structure of the server
    - can also use `@WebServlet(name="ClassName", urlPatterns={"/path1", "/path2"})` annotation

```
// whenever `/this_is_path` is called, it runs the code in the thing1 class
<servlet>
        <servlet-name>Internal name</servlet-name>
        <servlet-class>thing1</servlet-class>
        // optional, if 1 then initialize this servlet on server startup than first request
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>Internal name</servlet-name>
        <url-pattern>/this_is_path</url-pattern>
</servlet-mapping>
```

- **ServletContextListener**
    - Allows you to pass objects that can be accessed by servlets
    - `javax.servlet.ServletContextListener`
        - is created when the web application is created, before any of the servlets
    - Creating a `ServletContextListener`

```
@WebListener // make java autodiscover the class
public class ContextListener implements ServletContextListener{
        // called when the application is created
        public void contextInitialized(ServletContextEvent sce){
                // get the servlet context
                ServletContext context = sce.getServletContext();
                context.setAttribute("key", value);
        }

        // called when the application is stop. Needs to be overridden.
        public void contextDestroyed(ServletContextEvent e){
        }
}
```

- Getting and setting data
    - `getServletContext().getAttribute("key")` in the implementation
    - `sce.getServletContext().setAttribute("key", value)` in the `contextInitialized()` method
- you can add the `@WebListener` class annotation so Java autodiscovers the ServletContextListener that are creating, alternatively you can define it inside the web.xml file like request mapping

```
        // web.xml example
        <listener>
                <listener-class>path.to.class.Listener</listener>
        </listener>
```

Chapter 3

- Linking forms to servlets

    - the HTML form element has a `action` attribute which should point to the link of the servlet url pattern (remember to prefix it with the name of the deployment)

    ```
    <form action="<%= request.getContextPath() %>/name_of_servlet" method="POST">
            {your input fields here}
    </form>
    ```

    - it also has a method attribute which is `GET` or `POST`
        - dictates the type of request method to use and where to store the form parameters
            - GET - query string in the path
            - POST - hidden request body
- **Acquiring form data in servlets through the request parameter**
    - `request.getParameter(parameter_name)`
        - parameter name should match the `name` attribute of the input element you want to get
        - returns the first occurrence of `parameter_name` in the query string
            - will return an empty string if the parameter exists but has no value
        - returns null if it doesn't exist
            - For text inputs: you need to check that it isn't null before validating it
            - For checkbox inputs: it returns null if and only if it wasn't checked
    - `request.getParameterValues(parameter_name)`
        - returns an array of URL decoded values of all occurrences of `parameter_name` in the query string or the request body
        - returns null if `parameter_name` does not appear in query string or the request body
    - `request.getParameterNames()` and `request.getParameterMap()`
        - returns an `Enumeration<String>` of the parameter names and map of the request parameters respectively (`Map<String, String[]>`)

    ```
    // how to loop through all request parameter names
    for (Enumeration<String> e = request.getParameterNames(); e.hasMoreElements();)
            System.out.println(e.nextElement());

    // how to loop through all request parameters
    for(Map.Entry<String, String[]> entry : request.getParameterValues().entrySet())
            System.out.println(entry.getKey());
    ```

- Other methods of `HttpServletRequest`
    - `request.getCookies()` - returns an array of cookie objects of the request
    - `request.getSession()` - returns the `HttpSession` object associated with the user
        - An `HttpSession` object has `setAttribute()` and `getAttribute()` methods
    - `request.getMethod()` - returns a string that is the HTTP method of the request
    - `request.getHeader(header_name)` - returns the value of a header based on the name
        - `User-agent` - identifies the client category / device type and browser used
        - `Accept` - the mime types that the browser can handle
        - `Referer` - URL of the referring webpage for tracking traffic
        - `Cookie` - The saved cookies on the client
    - `request.getHeaderNames()` - returns an enumeration of the names of the headers of the request
- **`RequestDispatcher`**
    - used to forward the HTTP request to another URL
    - URL must be within the application
    - `request.getRequestDispatcher("/path/to/new.jsp")`
        - returns a `RequestDispatcher` object which has a `forward()` method that takes in the reqeust and response
        - absolute paths are redirected to `domain.com/domain_deployment_name/path`
        - `request.getAttribute("javax.servlet.forward.query_string")`

            - get the name of the servlet that forwarded to the current servlet
    - is used when request attributes must be rendered in the JSP file since `response.sendRedirect("/path")` sends a raw HTTP redirect instead of redirecting internally

Chapter 4

`response.setContentType()`

- set the content type header
- tells the browser what type of content you are sending back
    - same as the mime type
- You can only set the content type once for every response
    - Always set this first before you call the method that gives you the output stream

`response.getWriter()` - returns a PrintWriter which is the same interface as System.out
`response.getOutputStream()` - returns a `ServletOutputStream`

```
public void doGet(HttpServletRequest request, HttpServletResponse response){
        // set the content type so the browser knows what we're sending
        response.setContentType("application/jar");
        // get the file as an input stream relative to the deployment folder
        InputStream is = getServletContext().getResourceAsStream("/file.jar");

        // the number of bytes read from the input stream, is -1 if there are no bytes left
        int bytesRead = 0;
        byte[] one_kb = new byte[1024]; // 1KB of space
        OutputStream os = response.getOutputStream();

        // is.read(bytes) loads 1KB of the file into one_kb, and returns the number of bytes read. If the returned
value is -1, there are no more bytes to send
        while((bytesRead = is.read(one_kb)) != -1)
                // send one_kb of the file to the output stream, pass bytesRead so output stream knows when not all
of one_kb is populated
                os.write(one_kb, 0, bytesRead);

        os.flush(); os.close();
}
```

Setting response headers

- `response.setHeader("name", "header_value");` - sets an arbitrary header
    - `Content-Type`
        - The MIME type of the document being returned
        - `response.setContentType("text/html")` is equivalent to `response.setHeader("content-type", "text/html")`
    - `Refresh`
        - The number of seconds until browser should reload page
        - `requeset.setHeader("Refresh", "5; url=https://example.com/")`
            - Go to `https://example.com` after 5 seconds
    - `Cache-Control` - prevents the page from being cached
    - `Location` - use `response.sendRedirect()` instead
    - `Set-Cookie` - use `response.addCookie()` instead
- `response.addHeader("name", "header_value")`; - adds a new occurrence of the header
- `response.setDateHeader("name", milli_since_epoch)`;
- `response.addDateHeader("name", milli_since_epoch)`;
- `response.setIntHeader("name", intValue)`;
- `response.addIntHeader("name", intValue)`;

Sending status codes

- `response.setStatus(number)`
  - Status code constants are in `HttpServletResponse` and are preferred over using magic numbers
  - 200 - everything is fine - `HttpServletResponse.SC_OK`
  - 301 -requested document is temporarily moved elsewhere - `HttpServletResponse.SC_MOVED_TEMPORARILY`
    - **Redirect** - makes the client do the work
      - The new URL is seen in the browser
    - **Forward** - makes the server do the work
      - The client doesn't know some different resource is being sent
      - Forwards the request and response objects
  - 404 - content is not found - `HttpServletResponse.SC_NOT_FOUND`
- `response.sendError(code, message)` - wraps message inside a small HTML document

Chapter 5

- **WebApplications**
  - everything is bundled together in a single directory or file
  - access to content in the webapp is through a URL that has a common prefix
  - Many aspects of the WebApplication can be controlled through deployment descriptor
  - all compliant servers support web apps, the code can be redeployed on a new server by moving a single file or directory
  - **Registering a webapp**
    - Copy the `build/web` folder of deployment directory into Tomcat's webapps folder
    - Rename the web folder into the desired context path
  - **WAR Files**
    - A jar file with a different file extension
    - All servers are required to support Webapps in WAR files
    - Create: `jar cvf webAppName.war *`
    - In tomcat: drop the war file in `webapps` , the file name becomes the app name
- **Welcome file list**
  - accessible to all directories under your webapp, glassfish looks at the same list
  - uses the first thing that matches

```
<welcome-file-list>
        <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

- **Configuring error pages**

```
<error-page>
         // a catch-all error page
         <exception-type>java.lang.Throwable</exception-type>
         <location>/errorPage.jsp</location>
</error-page>
<error-page>
         <error-code>404</error-code>
         <location>/notFoundError.jsp</location>
</error-page>
```

- Loading a servlet on startup

```
<servlet>
        <servlet-name>My Servlet</servlet-name>
        <servlet-class>myServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
```

- `request.getContextPath()` or `${pageContext.request.contextPath}`
  - Allows you to use absolute paths without hard-coding the application prefix

  ```
  <img src="<%= request.getContextPath() %>/images/home.gif" />
  ```

Chapter 6

HTTP is a stateless protocol

- it provides no way for a server to recognize that a sequence of requests are all from the same client
- after every request, the connection between client and server is dropped and forgotten
- no memory between client connections

How to maintain state within HTTP

- **URL Rewriting**
  - Explicitly append the data you want to pass to the URL
  - Not advisable for lengthy and sensitive data since its seen in the URL
  - Need to encode the data into a URL-safe format using `URLEncoder.encode(testing, "UTF-8")`
- **Hidden fields**
  - Fields dynamically added to an HTML form that are not displayed in the client's browser
  - Used when you go the the next page through a form submission
- **Cookies**
  - Flow
    - Servlet sends a name and value to client
    - Client saves the name and value to the file system
    - The cookie is sent every time it connects to the same site
  - How to add a cookie

  ```
  // by default, a cookie disappears when the browser exits
  Cookie cookie = new Cookie("key", value);
  response.addCookie(cookie);
  ```

  - How to get cookies
    - `request.getCookies()`

  ```
  Cookie[] cookies = request.getCookies();
  for(Cookie cookie : cookies){
          cookie.getName(); // returns a string of the cookie's name
          cookie.getValue(); // returns a string of the cookie's value
  }

  Cookie newCookie = new Cookie("cookie_name", "cookie_value");
  // newCookie.setDomain(string)
  // newCookie.setMaxAge(number_of_seconds)
  // newCookie.setPath(string)
  // newCookie.setSecure(bool)
  // newCookie.setHttpOnly(bool)
  // newCookie.setValue()
  response.addCookie(newCookie);
  ```

  - How to remove a cookie

    - Get the cookie, set its age to 0 with the `setMaxAge(0)` method
    - Add the cookie to the response
- **Sessions**
  - Randomly generated session ID is used to identify a user session
  - SID is stored on the client as a cookie, Session data is stored on the server
  - `HttpSession s = request.getSession()`;
    - automatically creates the session and sends the session cookie in the response
  - `HttpSession s = request.getSession(false);`
    - don't automatically create a session
    - returns null if there is no preexisting session
  - **Setting and getting values in sessions**

```
session.setAttribute("uname", uname);
String str = (String) session.getAattribute("uname");
session.removeAttribute("uname")
```

  - `HttpSession` methods
    - `session.invalidate()` - when you are done
    - `session.getAttributeNames()` - returns names of all attributes in the session
    - `session.getId()` - returns the sid
    - `session.isNew()` - Determine if session is new to client.
    - `session.getCreationTime()` - return time when session was first created
    - `session.getLastAccessTime()` - return the last time the container got a request with this sid
    - `session.setMaxInactiveInterval(seconds)`
      - maximum time in seconds that client should send requests for this session
      - a session timeout of -1 means the session will never expire
      - **alternatively this can be set in the DD**

    ```
    <session-config>
            <session-timeout>value_in_minutes</session-timeout>
    </session-config>
    ```

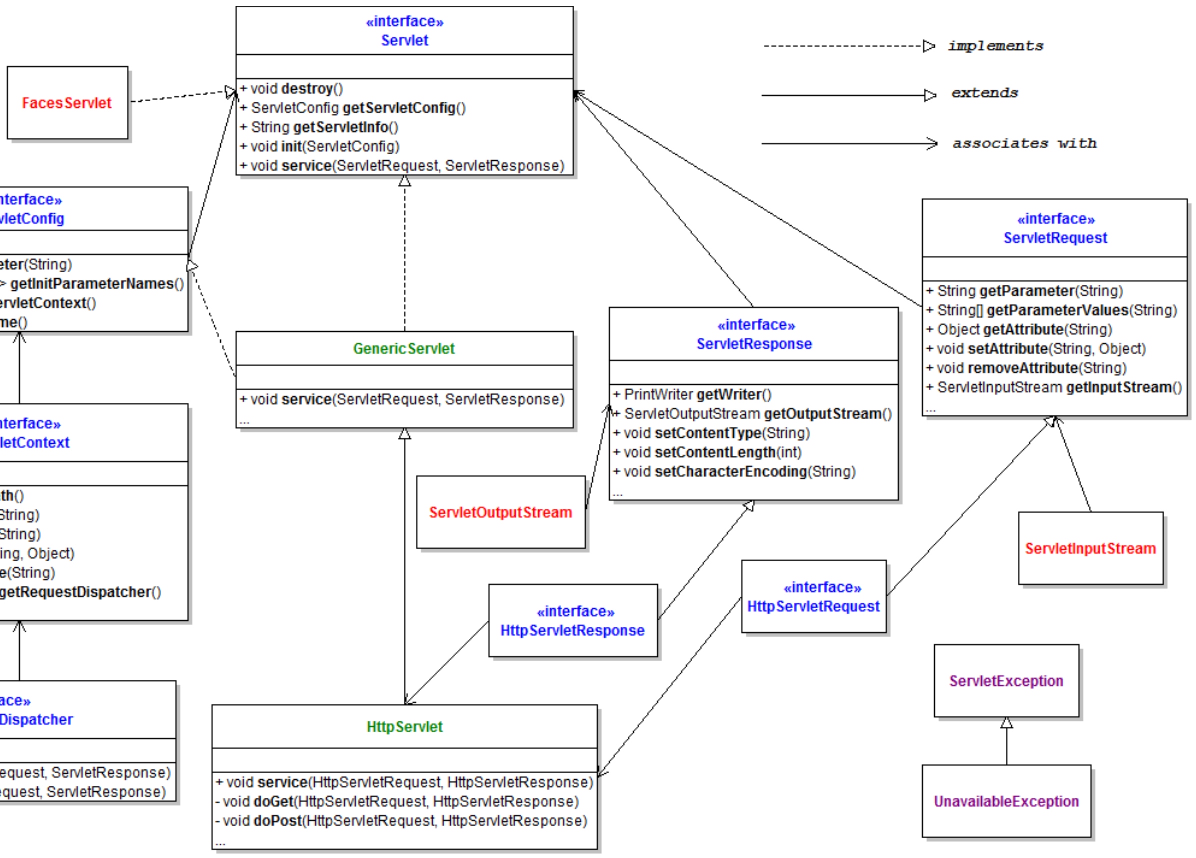
    - `session.getMaxInactiveInterval()`

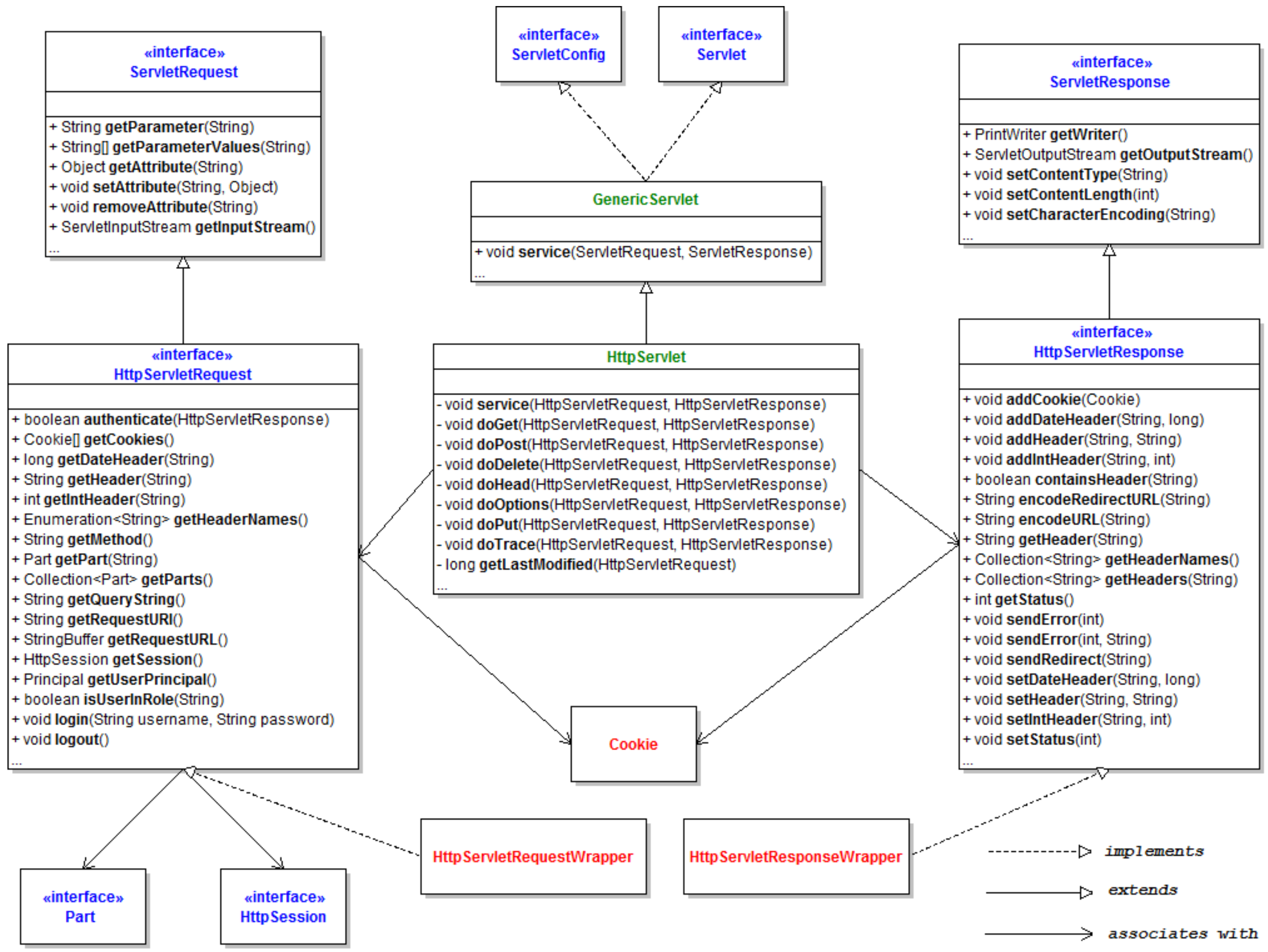
Chapter 7

JSP scripting elements

- The JSP is just a servlet
  - the container translates it into a java source file
  - the container compiles it into a java class
    - this class is reused and is recompiled whenever the JSP is modified
- **List of objects that you can use**
  - `request` - `HttpServletRequest`
  - `response` - `HttpServletResponse`
  - `out` - a buffered version of `JspWriter`
    - used to send output to the client
  - `session` - `HttpSession` associated with the request
    - unless disabled with session attribute of the page directive
  - `application`
    - The servlet context for sharing data
    - obtained through `getServletContext()`

    - `config`
        - Obtained through `getServletConfig()`
    - `exception` - only available through designated error pages
- Expressions: `<%= expression %>`
    - the expression is evaluated, converted into a String then placed in the html page
    - since its an expression, never end it with a semi colon
    - `<%= new java.util.Date() %>`
- Scriptlets: `<% script here %>`
    - Code that is inserted into the servlet's `_jspService()` method
        - meaning that any variables declared are reset on each page run
    - Not printed in the response, and can be used to add conditions to the page
- Declarations: `<%! code %>`
    - Code that is inserted into the servlet's class definition, outside of any existing methods
    - Can be used to declare variables and also methods.
- Directive: `<%@ page attribute="value" %>`
    - High level information about the servlet that will result from the JSP page
    - Can be used to import classes
        - `<%@ page import="foo.*" %>`
        - Is placed at the top of the generated servlet file
    - Specify the Mime type of the page generated
        - `<%@ page contentType="MIME-Type" %>`
    - Defines tag libraries available to the JSP
        - `<%@ taglib tagdir="path" prefix="cool" %>`
    - Defines text and code that gets added to the page at translation time
        - `<&@ include file="header.html" %>`
- Comments
    - HTML comments: `<!-- HTML Comment -->`
    - JSP comments: `<%-- JSP comment --%>`


implements
extends
associates with
«interface» Servlet
+ void destroy()
+ ServletConfig getServletConfig()
+ String getServletInfo()
+ void init(ServletConfig)
+ void service(ServletRequest, ServletResponse)
FacesServlet
«interface» ServletConfig
+ String getInitParameter(String)
+ Enumeration<String> getInitParameterNames()
+ ServletContext getServletContext()
+ String getServletName()
«interface» ServletContext
+ String getContextPath()
+ String getRealPath(String)
+ Object getAttribute(String)
+ void setAttribute(String, Object)
+ void removeAttribute(String)
+ RequestDispatcher getRequestDispatcher()
...
«interface» RequestDispatcher
+ void forward(ServletRequest, ServletResponse)
+ void include(ServletRequest, ServletResponse)
GenericServlet
+ void service(ServletRequest, ServletResponse)
...
«interface» ServletResponse
+ PrintWriter getWriter()
+ ServletOutputStream getOutputStream()
+ void setContentType(String)
+ void setContentLength(int)
+ void setCharacterEncoding(String)
...
«interface» ServletRequest
+ String getParameter(String)
+ String[] getParameterValues(String)
+ Object getAttribute(String)
+ void setAttribute(String, Object)
+ void removeAttribute(String)
+ ServletInputStream getInputStream()
...
ServletOutputStream
ServletInputStream
«interface» HttpServletResponse
«interface» HttpServletRequest
ServletException
UnavailableException
HttpServlet
+ void service(HttpServletRequest, HttpServletResponse)
- void doGet(HttpServletRequest, HttpServletResponse)
- void doPost(HttpServletRequest, HttpServletResponse)
...


«interface» ServletRequest
+ String getParameter(String)
+ String[] getParameterValues(String)
+ Object getAttribute(String)
+ void setAttribute(String, Object)
+ void removeAttribute(String)
+ ServletInputStream getInputStream()
...
«interface» ServletConfig
«interface» Servlet
GenericServlet
+ void service(ServletRequest, ServletResponse)
...
«interface» ServletResponse
+ PrintWriter getWriter()
+ ServletOutputStream getOutputStream()
+ void setContentType(String)
+ void setContentLength(int)
+ void setCharacterEncoding(String)
...
«interface» HttpServletRequest
+ boolean authenticate(HttpServletResponse)
+ Cookie[] getCookies()
+ long getDateHeader(String)
+ String getHeader(String)
+ int getIntHeader(String)
+ Enumeration<String> getHeaderNames()
+ String getMethod()
+ Part getPart(String)
+ Collection<Part> getParts()
+ String getQueryString()
+ String getRequestURI()
+ StringBuffer getRequestURL()
+ HttpSession getSession()
+ Principal getUserPrincipal()
+ boolean isUserInRole(String)
+ void login(String username, String password)
+ void logout()
...
HttpServlet
- void service(HttpServletRequest, HttpServletResponse)
- void doGet(HttpServletRequest, HttpServletResponse)
- void doPost(HttpServletRequest, HttpServletResponse)
- void doDelete(HttpServletRequest, HttpServletResponse)
- void doHead(HttpServletRequest, HttpServletResponse)
- void doOptions(HttpServletRequest, HttpServletResponse)
- void doPut(HttpServletRequest, HttpServletResponse)
- void doTrace(HttpServletRequest, HttpServletResponse)
- long getLastModified(HttpServletRequest)
...
«interface» HttpServletResponse
+ void addCookie(Cookie)
+ void addDateHeader(String, long)
+ void addHeader(String, String)
+ void addIntHeader(String, int)
+ boolean containsHeader(String)
+ String encodeRedirectURL(String)
+ String encodeURL(String)
+ String getHeader(String)
+ Collection<String> getHeaderNames()
+ Collection<String> getHeaders(String)
+ int getStatus()
+ void sendError(int)
+ void sendError(int, String)
+ void sendRedirect(String)
+ void setDateHeader(String, long)
+ void setHeader(String, String)
+ void setIntHeader(String, int)
+ void setStatus(int)
...
Cookie
HttpServletRequestWrapper
HttpServletResponseWrapper
«interface» Part
«interface» HttpSession
implements
extends
associates with

Servlet API

- **Servlet** - an interface that contains initialization / lifecycle methods
  - Methods
    - `init()` - executed once when the servlet is first loaded and before the servlet can service any client requests
      - a unique ServletConfig is created for the servlet
      - gives you a chance to init your servlet before handling any client requests
      - container reads the servlet init parameters from the Deployment Descriptor and gives them to servlet config, which is passed into the init method
    - `service()` - looks at the request and figures out what it should run
      - when a request is sent, the container starts a thread and invokes service().
      - service() looks at the request and invokes the matching `do<Method>()` post in the servlet
    - `destroy()` - called when the server deletes the servlet instance, not called after every request
      - cleaning resources
    - `getServletInfo()`
    - `getServletConfig()` - returns the config of the servlet from the `web.xml` file
  - **Subclass** - `GenericServlet`
    - An abstract class that implements the methods of the Servlet interface
      - It leaves the `service()` method unimplemented
    - Most of the secret behaviour lies in this class
      - **Subclass** - `HttpServlet`
        - an abstract class that implements the `service()` method, calling `do<MethodName>()` methods
        - This class is extended by your servlet
- `ServletRequest` interface
  - Methods
    - `getParameter(String name)` - returns a string from request parameters
    - `getParameterValues(String name)` - returns a `String[]` for parameters with multiple values
    - `getAttribute(String key)` - returns an Object from attribute map
    - `setAttribute(String key, Object value)` - sets an Object in the attribute map
    - `removeAttribute(String key)`
    - `getInputStream()`
    - `getRequestDispatcher(String relativePath)` - returns a RequestDispatcher object to the path
  - **Extended by** `HttpServletRequest` interface
    - Adds methods specific to HTTP, such as Cookies / Methods / Headers
      - `String getContextPath()` - returns the portion of the URI that indicates the context path of the application
      - `Cookie[] getCookie()` - returns an array of the request's cookies
      - `String getHeader(name), int getIntHeader(name), Enumeration<String> getHeaderNames()`
      - `String getMethod()`
      - `HttpSession getSession(), HttpSession getSession(boolean)`
        - returns an HttpSession object
- `ServletResponse` interface
  - Methods
    - `PrintWriter getWriter()`
    - `ServletOutputStream getOutputStream()`
    - `setContentType(String)`
    - `setContentLength(int)`
    - `setCharacterEncoding(String)`
  - **Extended by** `HttpServletResponse` interface
    - Adds methods for modifying cookies, headers,
      - `addCookie(Cookie)`
      - Raw methods to add headers
        - `addDateHeader(String key, long value)`
        - `setDateHeader(String key, long value)`

            - `addHeader(String key, String value)`
            - `setHeader(String key, String value)`
            - `addIntHeader(String key, int value)`
            - `setIntHeader(String key, int value)`
            - `boolean containsHeader(String)`
            - `String getHeader(String)`
            - `Collection<String> getHeaders(String key)`
                - returns an array of strings of the values for a key
            - `Collection<String> getHeaderNames()`
                - returns an array of strings of the key of the request headers
        - `int getStatus(), void setStatus(int)`
        - `void sendRedirect(String)`
            - send an absolute redirect relative to the domain of the client
- `ServletConfig` interface
    - Methods
        - `String getInitParameter(String)`
        - `Enumeration<String> getInitParameterNames()`
        - `String getServletName()`
        - `ServletContext getServletContext()` - returns the ServletContext, which is accessible by all servlets and JSPs
- `ServletContext` interface
    - Can also share objects between the entire application using `setAttribute()` methods.
    - `String getContextPath()` - get the context path of the application
    - `String getRealPath()`
    - Info sharing methods
        - `Object getAttribute(String key)`
        - `void setAttribute(String key, Object Value)`
        - `void removeAttribute(String)`
        - `String getInitParameter(String)`
        - `Enumeration<String> getInitParameterNames()`
        - `boolean setInitParameter()`
    - `RequestDispatcher getRequestDispatcher()` - get a RequestDispatcher instance
- `RequestDispatcher` interface
    - Allows you to forward a request to other servlets, making it seamless to the client
    - Methods
        - `void forward(HttpServletRequest request, HttpServletResponse response)`
            - Forward a request from a servlet to another resource
        - `void include(HttpServletRequest request, HttpServletResponse response)`
            - Include the content of a resource in the response
- **Cookie** class
    - `getName() / setName()`
    - `getValue() / setValue()`
    - `isHttpOnly() / setHttpOnly()`
    - `getDomain() / setDomain()`
    - `getMaxAge() / setMaxAge()`
        - Set a cookie's maxAge to 0, and add it to the response to delete the cookie from the frontend
    - `getPath() / setPath()`
    - `getSecure() / setSecure()`
- Listeners
    - `HttpSessionActivationListener` - notifies objects bound to sessions that sessions will be passivated and activated
        - A container that migrates sessions between VMs is required to notify all attributes that implement this interface

        - `void sessionDidActivate(HttpSessionEvent se)`
        - `void sessionWillPassivate(HttpSessionEvent se)`
    - `HttpSessionAttributeListener` - receives notifications about HttpSession attribute changes
        - is registered
        - `void attributeAdded(HttpSessionBindingEvent e)`
        - `void attributeRemoved(HttpSessionBindingEvent e)`
        - `void attributeReplaced(HttpSessionBindingEvent e)`
    - `HttpSessionBindingListener` - an object is notified when it is bound / unbound from a session (can be caused by a programmer unbinding the object, session being invalidated or a session timing out)
        - `void valueBound(HttpSessionBindingEvent event)`
        - `void valueUnbound(HttpSessionBindingEvent event)`
            - can tell you if the session is about to timeout
    - `HttpSessionIdListener` - notifications about HttpSession id changes
        - is registered
        - `void sessionIdChanged(HttpSessionEvent event, String old_id)`
    - `HttpSessionListener` - notifications about HttpSession lifecycle events
        - is registered
        - `void sessionCreated(HttpSessionEvent se)`
        - `void sessionDestroyed(HttpSessionEvent se)`
            - a session is about to be invalidated / timeout
    - `ServletContextAttributeListener` - notification events about ServletContext attribute changes
        - is registered
        - `void attributeAdded(ServletContextAttributeEvent e)`
        - `void attributeRemoved(ServletContextAttributeEvent e)`
        - `void attributeReplaced(ServletContextAttributeEvent e)`
    - `ServletContextListener` - receiving notification about ServletContext lifecycle events
        - `void contextDestroyed(ServletContextEvent sce)`
        - `void contextInitialized(ServletContextEvent sce)`
    - `ServletRequestAttributeListener` - notification events about ServletRequest attribute changes
        - is registered
        - `void attributeAdded(ServletRequestAttributeEvent e)`
        - `void attributeRemoved(ServletRequestAttributeEvent e)`
        - `void attributeReplaced(ServletRequestAttributeEvent e)`
    - `ServletRequestListener` - receiving notification about ServletRequest lifecycle events
        - `void requestDestroyed(ServletRequestEvent sre)`
        - `void requestInitialized(ServletRequestEvent sre)`

2609 Notes

Abstract Window Toolkit

- The package in the java API that allows us to create graphical user interface objects like buttons, frames, textareas and the like
- Superseded by the swing and JavaFX APIs
- Components
    - Buttons
    - Textfields
    - Label
- Containers
    - Panel
        - A type of container that provides a space

            - It cannot be launched itself, it needs to exist inside another container
        - Frame
            - Frame Class
                - A top level window with a border and title bar
                - Uses BorderLayout as the default layout manager
                - Has resizable corners
        - Dialog
- Layout Managers
    - Are used to position components inside containers
    - can be customized by nesting containers
    - From java.awt
        - FlowLayout
            - arranges components in a directional flow, like lines of text
            - used to arrange buttons in a panel, arranging them horizontally until no more buttons fit on the same line
            - the line arrangement is determined by the align property
                - LEFT, RIGHT, CENTER, LEADING, TRAILING
        - BorderLayout
            - lays out a container, and resizing components to fit in five regions, NORTH, SOUTH, EAST, WEST, CENTER
            - Each region can't contain more than 1 component
            - `(new Panel()).add(new Button("okey"), BorderLayout.SOUTH)`
        - GridLayout
            - lays out a container's components in a rectangular grid
            - container is decided into equal sized rectangles, and one component is placed in each rectangle, such that all components will still be equally sized
        - Cardlayout
        - GridBagLayout
    - javax.swing
        - BoxLayout
        - SpringLayout

Event Handling Techniques

- **Events** - objects that describe interactions between the user and a gui component
- **Event Sources** - GUI components that can be interacted with by the user
    - have methods that allow you to register event listeners with them. When something happens to the source, it sends a notification to all the listener objects for that event
    - All Information is encapsulated by an event object that is subclassed from `java.util.EventObject`
    - Different event sources can produce different kinds of events
- **Event handlers**
    - a method that is invoked every time that an event is called
    - where you put code that runs everytime an event occurs
- Techniques
    - Event delegation model
        - Allows you to delegate the event handler to a different class, producing a loosely coupled GUI and Event handler package
            - Each class is a separate Java source file
        - Can attach as many event handlers to one event source, just need to register each one to the source
    - Listeners - interfaces that you implement, you have to override all methods in the interface
    - Adapter classes - like listeners, but you're not required to override everything
        - only listeners that contain two or more methods have a corresponding adapter class
        - they have default implementations for the listener so you don't have to override everything

    - Inner classes
        - make for tightly coupled components, used to group the UI and event handler code tightly in a single codebase
    - Anonymous classes
        - creates an object on the fly that is anonymous
        - does not have any handler or object references
        - they are automatically destroyed, leaving memory in an optimized state
            - Are used usually for mobile devices / desktop applications that have an efficient runtime, since having less objects in memory improves runtime speed
    - Lambda expression
        - Allows you to pass unnamed functions as parameters to methods.
        - A simplified version of anonymous classes

Swing

- A part of the java foundation classes that is used to create window-based applications
- Built on top of AWT, but provides a set of platform independent and lightweight components
    - Swing is platform independent, AWT is platform dependent
    - Swing is lightweight, AWT is heavyweight
    - Swing look and feel can be customized, AWT cannot be customized
    - Swing provides mower powerful components, AWT provides less components
    - Swing follows MVC pattern, AWT does not
- `javax.swing` provides classes for the swing API:
    - JRadioButton - Implementation of a radiobutton, can be selected / deselected
        - Can be used in collaboration with a ButtonGroup to group buttons, so that only one can be selected in the group at a timer
    - JCheckbox - Implementation of a checkbox, an item that can be selected or deselected
        - Any number of checkbox in a group can be selected, and is toggleable by a click
    - JComboBox - a component that combines a button or editable field and a dropdown list
        - the user can select a value from the drop down list, which is then set as the default view, similar to `<select />`
        - If editable, the user can set their own values for it
    - JList - a component that displays a list of values, where the user can select one or more items
        - A separate model (ListModel) maintains the contents of the list, which must be created first before the JList itself
    - JButton
    - JTextField
    - JTextArea
    - JMenu
    - JColorChooser
- Converting from AWT to Swing
    - Add a package declaration at the top of the source code so it can be packaged in a jar file
    - Add J in front of the components and containers, but don't add it in the layout managers
- JAR file - Java Archive
    - A package file format typically used to group java classes into one file for distribution
    - Can be accessed and used in different operating systems
    - Without a manifest file:
        - Creating: `jar cvf <JarFile.jar> <list of files follows>`
        - Running: `java -cp <JarFile.jar> <package.name.class.with.Main>`
    - With manifest:
        - Creating: `jar cvfm <JarFile.jar> <Manifest.MF> <list of files follows>`
            - where Manifest.MF contains: `Main-Class: <package.name.class.with.Main>`
        - Running: `java -jar <JarFile.jar>`

Database

Database

- all data is stored in tables, made up of cols and rows
- Each table has one or more columns, each col has a datatype, and each row has a value for each column
- The data is structured in a particular way. A single item of data is stored in a name field
    - A complete set of fields makes up a record, the key contains data unique to that record.
    - All the records on one entity are stored in a table, and one or more tables then make up the database file

JDBC - Java Database Connectivity

- The API that manages connecting to a database, issuing queries and commands, and handling result sets obtained form the database
- Released as part of JDK1.1 in 1997, one of the first components developed for the Java persistence layer
- Available under `java.sql.*` and `javax.sql.*`
    - `java.sql.*` provides the API for accessing and processing data stored in a data source
    - `javax.sql.*` is the extensions used for the Java EE requirements
- the JDBC api allows to connect to multiple database sources under a single interface
    - You only need to change the database driver and how you establish the connection when changing from different databases
- Steps
    - Import Packages - `import java.sql.*;`
    - Load Driver - `Class.forName("org.apache.derby.jdbc.ClientDriver");`
    - Establish Connection
        - `Connection con = DriverManager.getConnection(connectionString, databaseName, password);`
        - Example connection string: `jdbc:derby://localhost:1527/FriendsDB`
            - the RDBMS to be used is part of the protocol
            - take note of the port where the RDBMS is running
            - and also the URI is the name of the database to connect to
    - Create and execute statement
        - `Statement stmt = con.createStatement();`
        - Can also use the PreparedStatement interface which precompiles SQL statements
    - Retrieve results
        - `ResultSet rs = stmt.executeQuery("SELECT * FROM PERSON_INFO");`
    - Close connection
        - `rs.close(); stmt.close(); con.close();`
- Difference between Statement and PreparedStatement
    - Statement
        - used for executing a static SQL statement in java JDBC
        - cannot accept parameters at runtime
        - is slower since it's building the query every time it's run
    - PreparedStatement
        - used for executing a pre-compiled sql statement
        - can be executed repeatedly with different with different parameters, therefore it's faster since it doesn't have to build the entire query every time it's run

MVC

Model View Controller

- Most webapps use the MVC design pattern where they separate the application logic from the user interface when designing software.

> - Has 3 layers
>   - Model - represents the business layer of the application as classes, the User table has a User class
>   - view - Defines the representation of the application (Web / Desktop)
>   - Controller - Manages the flow of data in the application
> - HTTP response codes - indicate whether an HTTP request has been successfully completed
> - sendRedirect vs RequestDispatcher
>   - `sendRedirect()` - a method of HttpServletResponse
>   - request is redirected to client and it will process the new URL
>   - Client can see on which page it has been redirected since it's done in the client side
>   - `RequestDispatcher`
>   - Can be accessed from HttpServletRequest
>   - Internally forwards the request to another servlet or JSP page
>   - The client doesn't know which page is processed internally, since processing is done in the server side