Java

Variable: Requires type declaration(cannot store any other type after) , naming(camelCase), value(static)

Primitive types: int,long(larger int), Boolean, double(float), char

String is not primitive – inherently an object

Wrapper Classes(uppercase first letter): more robust version that adds functionality to a primitive type – treated as an object

Objects stored in data structures – not primitive data types though

Conditionals:

If(condition) {}

* A condition must always evaluate to a boolean value, true or  false
* If the condition is true, it will execute the block of code between the curly braces { ... } , then skip any other else if or else statements in that chain and keep going.
* If the condition is false, it will skip that block of code { ... } but continue to the code immediately following that block which may be another else if or else statement.

Loops: We can use while/for interchangeably

While: We should use a while loop when we don't know beforehand how many times an operation must repeat, but we know we have to repeat until a certain condition has turned false.

For: We can think of a for loop as a specialized while loop that we should use if we know exactly how many times we have to iterate.

Strings methods:

Length/Concatenate/Format/indexof/trim/upper/lowercase/equality

.equals() instead of ==

Modularization:

One main method – the main method will run code in other files by making instances of those file’s classes – 1 imported in (simple import statement)

2 dependency injection(creating an instance of another class)

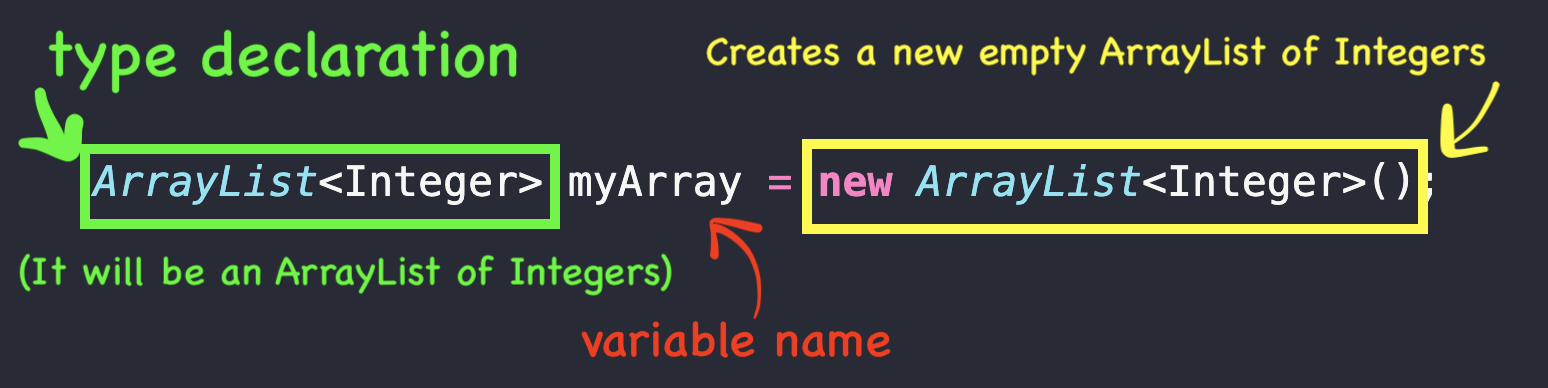
Type Casting: (switching types)

* Explicit – big to small
* Implicit – small to big

Fixed Arrays:

* Fixed size after initialization
* Array.length
* 0 based

Array List:

* Need to import
* Not fixed
* 0 based
* 
* Manipulate with .get / .set
* Add with .add

Maps:

* KV pairs are stored in maps
* Maps implemented in different ways
  + 1: HashMap
    - Store without order
* import java.util.HashMap;
* HashMap<String, String> userMap = new HashMap<String, String>();
* userMap.put("nninja@codingdojo.com", "Nancy Ninja");
* userMap.put("ssamurai@codingdojo.com", "Sam Samurai");
* String name = userMap.get("nninja@codingdojo.com");

Iterating over a hashmap:

Produce set of map’s keys using keyset() method of the HashMap class

Set: similar to h-m but has only values – all unique

import java.util.Set;

public class HashMapFun {

public static void main(String[] args) {

HashMap<String, String> userMap = new HashMap<String, String>();

userMap.put("nninja@codingdojo.com", "Nancy Ninja");

userMap.put("ssamurai@codingdojo.com", "Sam Samurai");

userMap.put("wwizard@codingdojo.com", "Walter Wizard");

// get the keys by using the keySet method

Set<String> keys = userMap.keySet();

for(String key : keys) {

System.out.println(key);

System.out.println(userMap.get(key));

}

}

}

Packages = folders

Naming convention = reverse domain name

OOP:

Object – collection of behaviors and properties that all revolve around the same concept

Classes – blueprints for object copying – named with noun / paschal case / class{}

Member Variables:

We can group data together as member variables inside a class to make our own custom data types

Getters Setters Access Modifiers:

1. The this keyword is necessary here to specify that we are talking about the instance variable and not the parameter variable.

Constructors and Method Overloading:

A **constructor** method is used to create new objects from class methods. The differences between a constructor method and the other methods are:

* It doesn't have a **return type**.
* The name of the constructor **MUST** be the same as the name of the class to which it belongs.
* A constructor is called automatically when a new instance of an object is created.
* what if we wanted to set the color of a vehicle to a default instead of calling the setter method after we created the object?

this:

Java, the this keyword refers to the current object within the context of an instance method or constructor.

There is one case in which you **MUST** use this: if one of your parameter variables shares the same name as one of your member variables. For example:

One of other uses of the this keyword is when you have overloaded your constructor, but don't want to have to write any repetitive code.

Object Superclass:

Top of class hierarchy

To use the methods have to use this

OOP:

Encapsulation(more to organize it to one) : protecting core values and variables from outside manipulation except through methods designed for that purpose

Inheritance: The ability for one class to inherit properties from another, “parent” class / “extend

Polymorphism: Overriding – annotation @Override : writing a method that has the same method signature and return type as parent method – runs the latter method only

Overloading: different method signature – both methods as unique

Abstraction(more towards the hiding part):

Hiding the details of a class that are not relevant to the user

Super: public class Human extends Mammal{

}

copy

In this case, Human would be a subclass of Mammal, and Mammal would be the superclass of Human.

Constructors: Option to include multiple constructors in class / can then control which constructor is used with the parameters

POJO: pain old java object

Java bean: object with certain rules

Models (classes into models that rep data from db)- in spring must be java beans

* Class that has a 0 arg constructor
* Member variables are private.

Abstract vs Interfaces

Abstract:

-blueprint for other classes

-cant create object off of Abstract class

-subclass that can be instantiated

* Abstract classes are very similar to interfaces but not static and final by default
* Used for prevention of ambiguous classes that should not be instantiated
* When an interface is implemented, a regular class must implement all methods from the interface. However, if an abstract class implements an interface, it can implement some methods and leave it to the subclass to implement the rest of the methods.

Consider using abstract classes if any of these statements apply to your situation:

* You want to share code among several closely related classes.
* You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
* You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Consider using interfaces if any of these statements apply to your situation:

* You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
* You want to specify the behavior of a particular data type, but aren't concerned about who implements its behavior.
* You want to take advantage of multiple inheritances of type.

Abstract Methods:

* Can have abstract methods in any of abstract classes
* Method might be different(make noise-meow/bark) to different animals – method don’t have to be abstract though for subclasses
* Don’t specify a body for the method – declare and end
* Child class- create implementation of method have to

Why do we need ac/am?:

* Enforce and organize what all subclass of main class has to have

Interface:

* Can achieve similar to multi class inheritance.
* INTERFACES ARE NOT CLASSES
* An interface states **behaviors** that a class must **implement**.(no limit)

**An interface is similar to a class in the following ways:**

* An interface can contain any number of methods.
* An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
* The bytecode of an interface appears in a .class file.
* Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

**However, an interface is different from a class in several ways, including:**

* You cannot instantiate an interface.
* An interface does not contain any constructors.
* The only fields that can appear in an interface must be both static and final, that is, constants that cannot change.
* An interface is not extended by a class; it is implemented by a class.
* An interface can extend multiple interfaces; a class can only extend from one class, but it can implement multiple interfaces.

**Using Session**

Again, just like with the Model model if you want to use session in any of your mapped handler methods, just include a parameter of type HttpSession session in the method signature.

Session.setAttribute/getAttribute – to use

**Flash – data that only persists across the next request**

To use our flash data, we have to inject the RedirectAttributes into our method, and add flash attributes to it. For example, we might inject RedirectAttributes to our createErrormethod to test sending an error back to our index method:

@RequestMapping("/createError")

public String flashMessages(RedirectAttributes redirectAttributes) {

redirectAttributes.addFlashAttribute("error", "A test error!");

return "redirect:/";

FORMS:

Action: route that handles what happens when the submit button is pressed

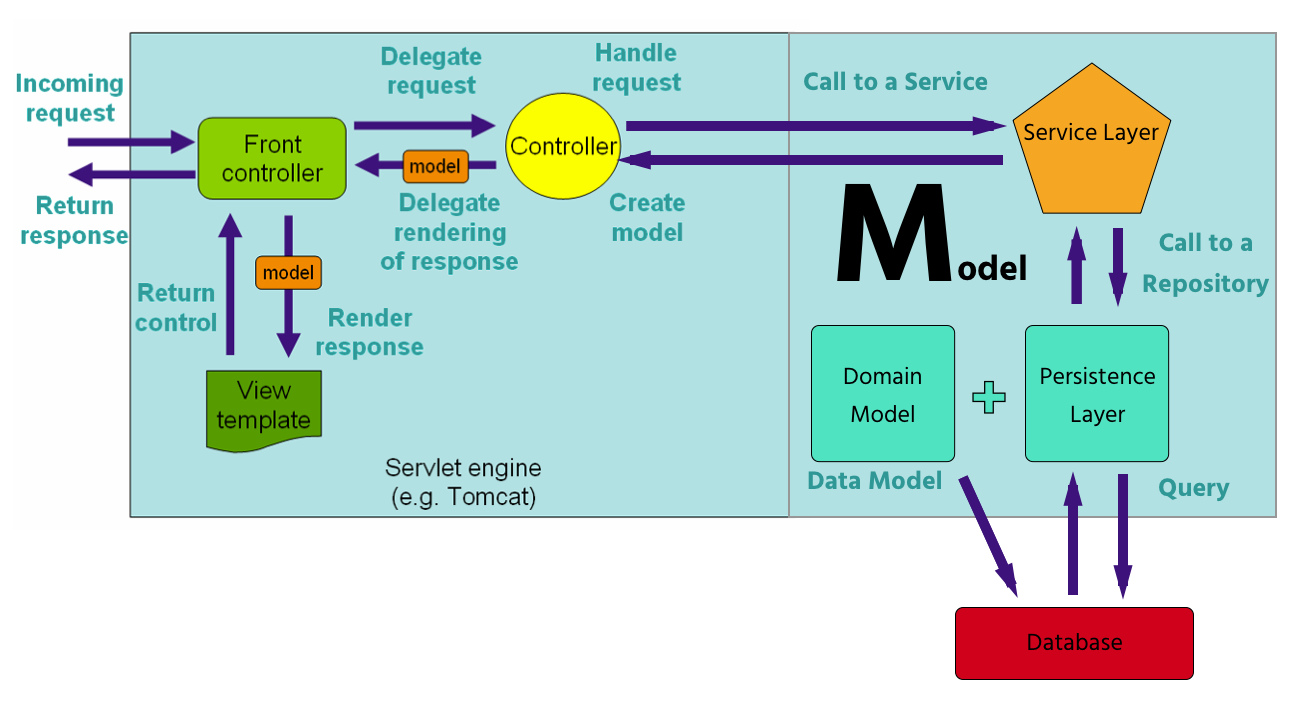
Shows vs process

Method: get/post/put

Input: input

Submit: <input type=”submit”> or <button>Submit</button> NEVER <input type=”button>

AVOID RENDERING ON A POST



* Persistence Layer: This layer is in charge of managing the application's data. PL is made up of Domain models and Repositories. We will use Repositories to access our database via an object relational mapper (ORM).
* Service Layer: It is made up of classes that implement the business logic of our application. It will call on the repositories to execute some sort of transaction according to the request from the user.

JPA- Java Persistence API – how to store data in a database

## **Adding Data Binding to the Controller**

Spring MVC has a process called ***data binding*** that allows for automatically writing the form data into a new object that we bind to the JSP page using the view model, that is, the Model model that you're accustomed to using.

@ModelAttribute

In the controller we'll use the @ModelAttribute annotation to handle our book creation and later updating as well. Here are the steps we'll be using to bind a book object to the view model:

1. Use the @ModelAttribute annotation to add an empty book object to the view model in the GET route that renders the form.
2. Render the view (new.jsp)
3. Fill the object with the form data including modelAttribute =””
4. Pass the filled book from the view model into the POST method
5. Save the new book to the database

ANNOTATIONS:

1: @RestController: Controller respond with JSON String Data

2: @RequestMapping: mapping web requests onto specific handler classes /methods

3: @RequestParam: ?id=1&name=john no mapping in URL but adding it to the url? Kind of

4:@PathVariable: /1/john

5: @Controller: Controller respond with view – pass data to view using a Model object (NOT OOP ONE)

6: @Repository

7: @Service

8: @Entity @Table

9:@Autowired

10: @Valid BindingResult result to display validations <form:errors path”/>

11: @Transient for confirm

11:  <input type="hidden" name="\_method" value="put">

<%@ page isErrorPage="true" %>  for put request

In Spring MVC, whenever you import Model into your controller you can use it as a container to store data and pass that data to the view engine. @REQUESTMAPPING

public class HomeController {

@RequestMapping("/")

public String index(Model model) {

model.addAttribute("dojoName", "Burbank");

return "index.jsp";

}

}

In the JSP

<c:out value="${dojoName}"/>

DELETE: use hidden input and implement destroy action in controller

LOGIN AND REGISTRATION:

User : first model is to create users in the database – NEVER STORE PW and CONFIRM PW

LoginUser: validating input from the login form

REGISTERATION FORM: modelAttribute = “newLogin”

Password Security:

Bcrypt = hashing algorithm

User object for reg/ LoginUser for login/ BindingResult for model level validations

Adding custom errors with rejectValue:

if(!newUser.getPassword().equals(newUser.getConfirm())) {

result.rejectValue("confirm", "Matches", "The Confirm Password must match Password!");

}

Does the user exist?

Optional<User> potentialUser = userRepo.findByEmail(newLogin.getEmail());

potentialUser.isPresent()

## **BCrypt (string,salt)**

1. To create a hash of the user's password to store in the database:

String hashed = BCrypt.hashpw(newUser.getPassword(), BCrypt.gensalt());copy

1. To check if a user's password is correct:
2. if(!BCrypt.checkpw(newLogin.getPassword(), user.getPassword())) {
3. result.rejectValue("password", "Matches", "Invalid Password!");

}

Many to Many relationships:

-two one-to-many relationships with an entity for the middle table

@ManytoMAany – both entities

@JsonIgnore

@JoinTable the middle table our entities will be mapped to

* + @JoinTable(name="categories\_products"): The name of the middle table.
  + joinColumns: @JoinColumn(name=productsThe foreign key that matches the primary key of the embedded class when the tables are joined.
  + inverseJoinColumns: @JoinColumn(name=cat)The foreign key that matched the foreign key of the opposite class when the tables are joined.