CS 1331 - Object-Oriented Programming

Jerry Chen

Contents

1 Exam 1

1.1 Java Introduction

1.2 Brief History

- Developed by James Gosling with the goal of creating a new programming language for consumer electronics like TVs.
  - Must be platform-independent. It needs to run on all sorts of hardware and operating systems.
- Originally named Oak. Sun Microsystems renamed it Java, and then it was acquired by Oracle.

1.3 Interpreted vs. Compiled

The flow for a compiled language goes: 1. Source code is written. 2. The compiler translates it to machine code, which looks different depending on the architecture/OS of the computer. 3. Linker introduces pre-compiled libraries (e.g., standard libraries). 4. Produces an executable file.

Interpreted languages execute line by line down the file as it's read.

1.4 The JVM

The Java Virtual Machine allows Java to be platform-independent. It's a software engine that produces a runtime environment for executing Java bytecode.

Java is both compiled and interpretd: 1. Write Java source code (.java file) 2. Compile it into bytecode (.class file) 3. As long as a JVM is available for a given platform, that same bytecode can be interpreted and executed, regardless of hardware or OS. - The Just-In-Time compilation identifies hotspots in the code (where the code is run frequently) and compiles that into more optimized native machine code (which may vary with OS). - Otherwise, the bytecode is executed by the JVM interpreter.

Primary Breakthrough: write once, run anywhere

1.5 Java Quirks

1.5.1 Class-Centric

- Cannot have code outside of classes.
- Cannot have methods within another method.
- Each `.java` file can contain at most one `public` class, which must match the name of the file.
- Top-level classes must either be public for package-private. Private classes must be contained within other classes.

1.5.2 Compiling and Running Bytecode

- The command to run the java compiler is: `javac FileName.java`.
  - A `.class` bytecode file is created.
- The `.class` file can be run by the JVM using: `java FileName` (no file extension necessary).

1.5.3 Distinctions from Other Languages

- No chained assignment: `x = y = z`.
- Boolean type in Java is represented by `boolean`.
- `else` statements always correspond to the nearest `if` (regardless of indentation), if there are no braces explicitly defining the correspondance.
- `String.replace(target, newVal)` method replaces all `target` values, returning a new string.

1.6 Variables, Types, and Operators

1.7 Variables

Variables are containers for storing data values. Identifiers are case-sensitive strings of characters used to name variables, methods, classes, etc. They can contain letters, digits, and underscores, but cannot start with a digit.

- Declaration: A statement that creates a variable but does not give it a value.
  - `float twoThirds;`
- Assignment: A statement that gives a previously declared variable a value.
  - `twoThirds = 2/3;`
- Initialization: A declaration and assignment statement put together.
  - `float twoThirds = 2/3;`

1.8 Naming Conventions

In Java, `camelCase` is used for variable and method names and `PascalCase` is used for class and file names.

1.9 Primitive Types

- `byte`: 1 byte, stores whole numbers from -128 to 127
- `short`: 2 bytes, stores whole numbers from -32,768 to 32,767
- `int`: 4 bytes, stores whole numbers from -2,147,483,648 to 2,147,483,647
- `long`: 8 bytes, stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- `float`: 4 bytes, stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
- `double`: 8 bytes, stores fractional numbers. Sufficient for storing 15 decimal digits
- `boolean`: 1 bit, stores `true` or `false` values
- `char`: 2 bytes, stores a single character/letter or ASCII values

Note: `null` values cannot be assigned to variables declared to be of a primitive type. This only works for reference types, which are all arrays and classes/wrapper classes (basically all things except for primitive types).

1.10 Literal Numeric Values and Type Suffixes

To specify the value of a numeric literal in Java, there are 3 different type suffixes. By default, the `double` value is used for floating point numbers.

This causes issues during initialization, where any decimal number assigned to a float would cause an error.

```
float myFloat = 1.0f; // without `f`, there is an error due to lossy conversion from double to float
double myDouble = 1.0d; // optional, because this is default behavior
long myLong = 1.0L; // only an error if assigned number is too large for `int` type

// note that for longs, assigning an overly large number to a variable of an `int` type only causes an
    overflow (wraps to min value) and not an error
```

- There is no difference between lowercase and capital letters used, but capital `L` is preferred for readability for longs.

1.10.1 Type Casting

- Widening Cast (automatic): `byte` -> `short` -> `char` -> `int` -> `long` -> `float` -> `double`

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double
System.out.println(myInt);      // Outputs 9
System.out.println(myDouble);   // Outputs 9.0
```

- Narrowing Cast (manual, needs cast): `double` -> `float` -> `long` -> `int` -> `char` -> `short` -> `byte`

```
double myDouble = 9.78d;
int myInt = (int) myDouble; // Manual casting: double to int
System.out.println(myDouble);   // Outputs 9.78
System.out.println(myInt);      // Outputs 9 (data loss)
```

1.11 Modifiers

Modifiers are all of the keywords that appear in a class/method header. They are important for determining the scope of how said class/method is accessed, as well as providing additional properties or features. - They are not part of the function signature (which is only name + parameters).

1.11.1 Access Modifiers

There are four types of access modifiers:

Notes: - A top-level class can only be `public` or package-private (default).

1.11.2 Non-Access Modifiers

1.11.2.1 **`static`** Methods: - Belongs to the class level and must be accessed through `ClassName.staticMethod()`. - Static methods cannot access (read or modify) any instance fields, since it only has class-level info.

Fields: - Both static and normal methods can modify static fields, and this change is reflected across all instances.

1.11.2.2 **`final`** Classes: - Cannot be extended or inherited by any other class.

Methods: - Cannot be overriden by any subclass. Its behavior is consistent across all instances.

Fields: - Ensures that the object being referenced by the variable never changes (e.g., something else can never be assigned to this variable name). - If the object is a primitive type, this guarantees constant value. - If the object is a reference type (e.g., array), then the values within that array may still change but that object will be tied to this variable name. - A final field must be provided a value upon declaration OR in all reachable constructors of the class.

1.11.2.3 **`abstract`** Denotes abstract class/method

1.12 Operators

1.12.1 Arithmetic

- + (addition), – (subtraction), * (multiplication), / (division), % (remainder)

Note: Modulo may be negative if the divided is negative, unlike Python. For example, `-7 % 3 == -1` , not 2. - Remainder matches dividend sign; modulo matches divisor's sign.

This is because in Java, / results in integer division, which truncates. In Python, it's combined with floor division.

1.12.2 Operator Precedence Table

Precedence	Operator Type	Operators
1 (Highest)	Postfix	`expr++`, `expr--`
2	Unary	`(type-casting)`, `++expr`, `--expr`, `+expr`, `-expr`, `~`, `!`
3	Multiplicative	*, /, %
4	Additive	+, –
5	Shift	<<, >>, >>>
6	Relational	<, >, <=, >=, `instanceof`
7	Equality	==, !=
8	Bitwise AND	&
9	Bitwise exclusive OR	^
10	Bitwise inclusive OR	|
11	Logical AND	&&
12	Logical OR	||
13	Ternary	`? :`
14	Assignment	=, +=, -=, *=, /=, %=, &=, ^=,
		| | | Watch out for this type of question: ```java int a = 9;
		`int b = 2; double x = (double) a / b;` ``` –(double)
		`aacts beforea` / b‘, so the result is 4.5

1.12.3 Ternary Operator

The syntax is: `condition ? value_if_true : value_if_false;`

An example would be `int x = (y % 2 == 0) ? 2 : 1`, which would be equivalent to:

```
int x;
if (y % 2 == 0) {
    x = 2;
} else {
    x = 1;
}
```

1.13 Program Control Flow & Iteration

1.14 If-Else Statement

```
if (condition1) {
    // do a
} else if (condition2) {
    // do b
} else {
    // do c
}
```

Note: The Java compiler does not read indents or new lines. Instead, semicolons are used as separators between statements, and tabs are entirely used for formatting.

Importantly, an "else" statement always matches with the nearest "if" statement regardless of indentation level if you do not use braces to separate statement groupings.

1.15 Switch Statement

When checking the value of an expression against a list of potential values, use the switch syntax:

```
switch (grade) {
    case 'A':
        comment = "Excellent!";
        break;
    case 'B':
        ...
    default:
        comment = "Invalid";
        break;
}
```

Important Notes: - Without `break`, regardless of case, the code in the blocks below will execute once a case matches. - `default` is only considered if no other `case` matches. It is always last. - However, if the `default` block is executed and does not have a `break`, then it will still have fall-through behavior (if it's ever placed above other `case` statements. - There cannot be duplicate cases in terms of the value matched.

1.16 While Loops

Used to repeat a block of code 0 or more times:

```
while (expression) {
    // insert code here
}
```

A do-while loop is guaranteed to run at least once:

```
do {
    // something
} while (expression)
```

1.16.1 For Loops

Used to iterate 0 or more times, usually when the number of iterations is known.

```
for (int i = initialValue; i < endValue; endAction) {
    // Note that the condition for continuation is evauated **before** the loop runs
    // Insert code here
}
```

For something like looping from a range of 0 to 3 (3 times), the code would look like:

```
for (int i = 0; i < 3; i++) {
    // something
}
```

To iterate through the items in a collection:

```
for (dataType item : collection) {
    // something
}
```

- Requires datatype!!!

For non-list collections like `HashSet` and `HashMap`, use the `Iterator` interface, which allows for traversal of a collection and removal of elements during iteration (something that cannot be done with forEach).

```
Iterator<dataType> iterator = someCollection.iterator()
while (iterator.hasNext()) {
    dataType element = iterator.next()
    if (element.equalsIgnoreCase("target")) {
        iterator.remove() // removes the current element (that is, iterator.next())
    }
}
```

1.17 Core Java APIs

1.18 Classpath and Imports

1.18.1 Methods of Importing Packages

There are three ways to import or include a package in the code: 1. Explicit imports: `import java.util.Random` - Writing `import java.util.Random` tells Java that whenever `Random` is called within the code, it means `java.util.Random`. 2. Wildcard imports: `import java.util` $*$ 3. Full-Path: `java.util.Random random = new java.util.Random()`

1.18.2 Built-In Packages

When installing JDK, standard built-in packages like `java.lang`, `java.io`, and `java.util` are included within the JDK installation directory. A package, or namespace, is a way to organize related classes and interfaces together.

The JVM is able to locate the standard class libraries through the system's classpath. After installing the JDK, it's automatically configured to include where the built-in packages are, so when an import statement is used in the code the JVM can fine it.

1.18.3 User-Created Packages

When using classes from user-created packages, the class path needs to be specified. The default class path, if none is specified, is the current working directory.

First, the package must be declared in the very first line of the source code in the file containing the imported contents. The declaration must match the directory path relative to the classpath root.

```
// Within ./package1/test/Library.java
package package1.test;

public class Library {
    // content
}
```

Then, import the class

```
import package1.test.Library // import specific  class

// rest of code here
```

For something where the source code and packages are in separate folders, navigate to the `project_folder` directory and use the classpath flag while running: - `javac -cp "packages:src/main" src/main/Test.java` - `java -cp "packages:src/main" main.Test`

The `-cp` flag tells Java a list of separate locations to search for imported classes at. The colon is the path separator (on Windows it's a semicolon).

Then, when Java sees a statement like `import com.mycompany.utils.Calculator;`, it knows to search inside `packages` first.

```
project_folder/├──
 src│
   └── main│
       └── Test.java└──
 packages
    └── com
        └── mycompany
            └── utils
                └── Calculator.java
```

1.19 Strings

1.19.1 Definition and Properties

A *String* in Java is an object which represents a sequence of characters. Internally, the structure is a `private final char[]` array.

Strings are immutable. This means that for string literals, the JVM uses a memory structure called the string pool. When a string literal is encountered in the source code, the JVM checks that pool to see if there is already a string object with the same value. If there is, the JVM can assign a reference to the existing object instead of creating a new one. This process is called interning. - For string objects created using the new keyword, JVM uses a memory structure called the heap. These strings are not interned.

To define a String object:

```
String string = "Java";
String string = new String ("Java");
String string = "Hi" + " " + "world!";
```

- Strings must be enclosed in double quotes.
- The + operator is overloaded for string concatenation: `s1 + s2 == s1.concat(s2)`.

1.19.2 Common Methods

- `.length()`
- `.equals(Object anObject)`
- `.equalsIgnoreCase(String anotherString)`
- `.substring(int beginIndex, int endIndex)`
- `.charAt(int index)`
- `.isEmpty()`
- `.indexOf(String str)`
- `.toUpperCase()`
- `.toLowerCase()`
- `.compareTo(String str)`
  - Returns comparison (negative integer if smaller, 0 if equal, positive integer if larger).

1.19.3 Conversions

- String to Primitive: Use wrapper class `parse` methods.

```
String strInt = "123";
int numInt = Integer.parseInt(strInt); // numInt will be 123

String strDouble = "123.45";
double numDouble = Double.parseDouble(strDouble); // numDouble will be 123.45
```

- Primitive to String: Use `String.valueOf()` or concatenate with an empty string.

```
int num = 123;
String strNum1 = String.valueOf(num); // strNum1 will be "123"
String strNum2 = num + ""; // strNum2 will also be "123"
```

1.19.4 Format Specifiers for **`printf()`** Statements

The general form is `%[width].[precision]<format-specifier>`. Common format specifiers include: - `%s` for strings: width specifies min size of field (to be filled with padding) and precision specifies how many characters to print from the string. - With padding, text is usually right-aligned (padding on left). Adding a – right after the `%` allows for padding to go on the right. - `%d` for decimal representations of integers; precision is not supported. - `%f` for floating point numbers (e.g., float, double); width = # of characters, precision = # of decimal places.

Note that the precision is always applied first, and then width.

To pad with leading zeros instead of spaces, insert `0` right after `%` sign.

1.20 StringBuilder Object

1.20.1 Definition and Properties

A StringBuilder object is essentially a mutable string. It is implemented internally as a dynamic resizable `char[]` array.

It is just slightly less optimal than String in terms of memory, but it is more optimal when multiple changes are expected to be implemented on a string in a single-threaded context.

1.20.2 Common Methods

Informational Methods: - `.substring(int start, [int end])` - `.charAt(int index)` - `.indexOf(String str, [int fromIndex])`: returns index of first occurrence of target match, with optional start specification - `.lastIndexOf(String str, [int fromIndex])`

Modification Methods: - `.append(String str)` - `.insert(int offset, String str)` - `.setCharAt(int index, char ch)` - `.delete(int start, int end)` - `.deleteCharAt(int index)` - `.reverse()`: reverses `StringBuilder` in-place

Transformations: - `.toString()`

1.21 Array Objects

1.21.1 Definition and Properties

An array is a data structure in Java which stores a fixed number of elements of the same type.

Notably, it can contain both primitive types and reference objects. It can pretty much store anything, though its core limitation is that its size is fixed.

It can be declared/initialized in a few ways:

```
int[] intArray = new int[5]; // array with default values of specified size
int[] intArray = new int[]{1, 2, 3, 4, 5}; // declare + initialize
int[] intArray = {1, 2, 3, 4, 5}; // shorthand method
```

They are declared like this:

```
dataType[] arrayReference;

// Alternative (but not preferred) declaration
dataType arrayReference[];
```

Creation of array object and assignment of values in one step:

```
// Long way
int[] myInts = new int[] {1, 2, 3};

// Shorthand (array initializer)
int[] myInts = {1, 2, 3};
```

In separate steps:

```
// Either declare only the reference variable and then assign
int[] numbers;

// Allow compiler to infer size
numbers = new int[] {1, 2, 3};
// Or match size with number of values (error occurs if length doesn't match)
numbers = new int[5] {1, 2, 3, 4, 5};

// OR declare reference variable and assign it the address of created array
double[] arrayRefVar = new double[10];
arrayRefVar[2] = 5.6; // and so on...
```

- `new int` with specified size MUST be used, or an error will occur, unlike using the shorthand array initializer approach.

1.21.2 Accessing Values

Access values via bracket notation: `arr[1] = 5`. - Zero-indexed - Throws `ArrayIndexOutOfBoundsException` if index is out of bounds

Pass By Reference When an array is passed as an argument to a function, it is by reference (that is, a copy of the memory address of the array is given). - Therefore, any changes made to specific indices of the array within that function are also reflected. It is mutable.

Field The only field associated with the default array object in java is length (public & final). - `.length`: obtain fixed size of array

1.22 `java.lang`

The `java.lang` package is automatically imported into every Java file. It includes wrapper classes, the string class, StringBuilder, core classes like Object and System and Exception, and finally the Math class.

1.22.1 `java.lang.Math`

The Math class is entirely stateless (entirely constants and static methods). Most methods in `java.lang.Math` are overloaded to work with any numerical type.

Fields/Constants - `Math.PI`: a double that is closer than any other to π - `Math.E`: a double that is closer than any other to e

Math Operations - `Math.pow(base, exp)` - `Math.sqrt(num)` - `Math.cbrt(num)`

Random - `Math.random()`: return double between `[0.0, 1)`

Statistical Operations - `Math.abs()` - `Math.min()`: return min of 2 arguments - `Math.max()`: return max of 2 arguments

Rounding and Conversions - `Math.round()`: return closest int/long; half always rounds up - `Math.ceil()`: return smallest integer greater than argument - `Math.floor()`: return largest integer smaller than argument - `Math.toDegrees()`: radians –> degrees - `Math.toRadians()`: degrees –> radians

1.23 java.util

1.23.1 java.util.Random

To use the random object, a new instance must be created. Unlike `java.lang.Math`, this class does not have any static methods.

```
import java.util.Random;
// ...
Random random = new Random(); // or new Random(42)
```

- Optionally, the class can have a seed, which fixes the sequence of pseudo-random numbers generated to be the same.

The following generate random numbers over a uniform distribution: - `random.nextInt()`: any integer, + or - - `random.nextInt(someBound)`: integer from `[0, someBound)` - `random.nextBoolean()`: some boolean - `random.nextFloat()`: float from `[0.0f, 1.0f)` - `random.nextDouble()`: double from `[0.0d, 1.0d)` - `random.nextLong()`: any long number

1.23.2 java.util.Scanner

To use the Scanner object, a new instance must be created.

```
import java.util.Scanner
// ...
Scanner scanner = new Scanner(System.in); // InputStream object
// ...
scanner.close(); // Good practice
```

Useful Methods: - `scanner.hasNext()`: returns a boolean for whether there is a next token; used before `scanner.next()` to avoid any errors - `scanner.next()`: returns the next complete token as a string; tokens are delimited by any whitespace - `scanner.nextLine()`: reads an entire line and stops at newline character, consuming it in the process but not returning it

Core Reading Methods: The `java.util.Scanner` object has instance methods for all primitive data types in Java except for `char`. - To get the next `char`, you have to use `scanner.next().charAt(0)`.

1.23.3 java.util.ArrayList

1.24 java.io

1.25 Classes, Objects, and Memory

1.26 Classes and Objects Relationship

- Class: A blueprint for creating objects.
- Object: An instance of a class.

```
public class MyClass {
    int x = 5;

    public static void main(String[] args) {
```

```
        MyClass myObj = new MyClass();
        System.out.println(myObj.x);
    }
}
```

By default, class-level and instance-level variables will be assigned a value. The JVM assigns this before any constructors are run, so that all fields / instance variables will have a default value: - `boolean` defaults to `false` - `int` and other numbers (e.g., `float`, `double`) default to `0` - `char` defaults to `'\u0000'` (null char placeholder) - `String` and other objects default to `null`

This ALSO means that block-level variables will not be assigned a value. They need to be initialized before they can be used at all.

1.27 Wrapper Classes

Wrapper classes use an object to wrap around a primitive data type. - Autoboxing: Automatic conversion of primitive types to the object of their corresponding wrapper class. - Unboxing: The reverse of autoboxing.

For example, `int` vs. `Integer`: - `int`: Primitive data type, faster and more memory-efficient, no methods. - `Integer`: Wrapper class, an object that wraps an `int`, necessary for Collections Framework, provides utility methods (e.g., `parseInt`, `toString`).

Consider autoboxing as placing a value into a container (object), and unboxing as taking the value out of the container. ### Autoboxing and Unboxing Modern Java automatically converts between `int` and `Integer`.

```
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(10); // Autoboxing: 10 (int) is converted to new Integer(10)
int firstNum = numbers.get(0); // Unboxing: The Integer object is converted back to an int
```

Autoboxing is when the primitive value is converted to its corresponding wrapper class object. This happens during assignment and during method invocation (to ensure compatible types).

Unboxing is the automatic conversion of a wrapper class object back to a primitive type. It occurs for the above and also to support arithmetic operations.

1.28 Enum Class

An enum (or enumeration) object is used to assign a constant and descriptive name to any data type; most commonly, integers.

That way, instead of typing out:

```
if (statusCode == 1) {
    // there was an error
} else if
```

Enums are a special type of class that can hold individual constants. Each constant defined within an enum is then an instance of that enum class (considered objects, and inherit from `java.lang.Object`.

```
public enum Weekdays {
    MONDAY;
    TUESDAY;
    WEDNESDAY;
    THURSDAY;
    FRIDAY;
}
```

- Every single object within this enum class is then declared internally as `public static final`:
  - `public` means that any class which has access to the Enum class also has access to these instances.
  - `static` so it belongs to the enum class itself and not any instance of the class; accessible via `Weekdays.MONDAY`

  - `final` so that the reference to the object is a constant and cannot be reassigned to point to another object. Its state is also immutable.

Enums as a Class Since an enum is a special kind of class, it can also have fields (variables), constructors, and methods.

```
public enum CoffeeSize {
    SMALL("Small cup", 8); // call constructor for each enum instance
    MEDIUM("Medium cup", 10);
    LARGE("Large cup", 12);

    // Must declare instance variables (attributes) first
    // Standard is `private final` so that only instances can access
    private final String cupSize;
    private final int ounces;

    CoffeeSize(String cupSize, int ounces) {
        this.cupSize = cupSize;
        this.ounces = ounces;
    }

    // Define methods for the Enum
    public int getSize() {
        return this.cupSize;
    }

    // Alternatively public static, but then you cannot reference `this` within function body
}
```

- All instance variables which appear within the parameter list of the constructor must be declared beforehand. Convention is above the constructor, but below won't cause errors.
  - This is required because access modifiers cannot be specified within constructor function signature.

Built-In Enum Methods Static Methods: - `.values()`: returns array of enum constants in order of declaration - `.valueOf()`: convert string to actual enum object; case-sensitive and throws errors

Called on an instance: - `.name()`: returns name of the enum constant as a string; useful for unknown enum variables - `.ordinal()`: returns the numeric zero-based index of enum, using declaration order

1.29 Memory Allocation

For our running program, there are four areas of memory allocated to it. The first is to store the actual source code. The second is to store the initialized and declared global variables. The third is the stack, which is a last-in-first-out structure managing all function calls and the local variables and parameters within those functions. And the fourth is the heap, which is used for dynamic memory allocation with keywords like new.

1.30 Ambiguous Invocation

Method overloading occurs when two methods of the same name but accepting different arguments are created. - Other parts of method declaration, such as access modifiers and the `static` keyword, are not part of the function signature, and therefore cannot be used to overload methods (–> results in error).

Ambiguous invocation is a compile-time error that occurs when a function does not receive arguments perfectly conforming to the parameter types, but also does not know how to promote properly.

An example of this is defining `max(double a, int b)` and `max(int a, double b)`, and then saying `max(1, 2)`. Java doesn't know which one to promote to a wider form (int –> double), since both may be equally valid, so it throws an error.

2 Exam 2

2.1 Classes and Constructors

Classes can have fields/attributes (variables with values that may or may not differ among instances of the class), as well as methods.

All classes need a constructor, which is what is called during the creation of an object from a class. - If the user does not write one explicitly, the JVM provides a default constructor with no arguments that has the same access modifier as the class.

2.2 Defining Classes

A class is defined like this:

```
public class SomeClass {
    // Class-level variables (shared across all instances)
    public String className = "SomeClass"

    // Instance-level variables
    public String instanceName; // assign in constructor

    // Constructor
    // Cannot be more permissive than class's access modifier
    // Cannot be `static`; doesn't belong to the class itself
    public SomeClass(String instanceName) {
        this.instanceName = instanceName;
    }
}
```

2.3 A Note on Field Shadowing

Sometimes, a method will need to interact with an instance variable while also taking in a parameter, and these two variable names may overlap.

A good example of this is in a constructor:

```
public class Person {
    String name;

    public Person (String name) {
        // "this" is used to distinguish between field and parameter
        this.name = name;
        name = name; // does nothing; reassigns argument to itself
    }
}
```

It's good practice but not required if the parameter name is different from the instance variable name, but it must be used if they are the same.

2.4 Constructors

2.4.1 Constructor Chaining

Sometimes, you will want to offer a few different ways to construct an object using a class by passing in different arguments. This leads to the overloading of constructors: having multiple constructors with the same name but different parameter lists.

2.4.1.1 Within the Same Class Often, you can reuse previously-written constructor code. If there are multiple constructors within the same class taking in a different number of arguments (i.e., some provide a default value for a field if not provided), then constructors can call upon each other.

```
public class Book {
    private String title;
    private String author;
    private int publicationYear;

    // CONSTRUCTOR #1: Simplest constructor (fewest arguments).
    public Book(String title) {
        // Calls Constructor #2 below, providing a default author.
        this(title, "Unknown Author");
        System.out.println("-> Running the constructor with 1 argument...");
    }

    // CONSTRUCTOR #2: Takes title and author.
    public Book(String title, String author) {
        // Calls Constructor #3 below, providing a default year.
        this(title, author, 0);
        System.out.println("-> Running the constructor with 2 arguments...");
    }

    // CONSTRUCTOR #3: The main, "designated" constructor.
    // It does all the work.
    public Book(String title, String author, int year) {
        System.out.println("-> Running the constructor with 3 arguments...");
        this.title = title;
        this.author = author;
        this.publicationYear = year;
    }
}
```

- By convention, the constructor with the largest number of parameters is placed at the top.

2.4.1.2 With a Superclass For a subclass, a portion of the arguments passed in may be passed up to the superclass to be initialized using `super(<args>)`.

```
public class Person {
    protected String name;

    public Person(String name) {
        this.name = name;
    }
}
```

```
public class Employee extends Person {
    private int employeeId;

    public Employee(String name, int employeeId) {
        // 1. Pass the 'name' up to the Person constructor
        super(name);

        // 2. Now, handle the Employee-specific stuff
        this.employeeId = employeeId;
    }
}
```

- If the superclass has a no-argument constructor, and there is no `super()` or `this()` in the first line of the constructor, then the compiler automatically inserts `super()`.

2.5 The Base **`Object`** Class

`java.lang.Object` is the root of the class hierarchy. Every class has Object as a superclass.

2.5.1 Core Methods

- `equals(Object obj)`: indicates whether some other object is "equal to" this one.
  - By default, it is the most discriminating comparison possible. It is essentially `return (x == y)`, where it returns true if and only if `x` and `y` point to the same object in memory.
- `toString()`: returns a string representation of the object; called by default when attempting to print an object.
  - By default, returns `<className>@<hexadecimalHexCode>`.
- `hashCode()`: returns an integer hash code value for the object.
  - Make sure to override this function whenever equals is overridden, to ensure that two "equal" objects always return the same hash code.

2.5.2 Overriding Default Methods

`.equals()` This often needs to be overriden for more specific implementations of classes. For example, often you don't want to compare objects by location in memory, but rather whether a set of internal fields are the same.

To do so, there is a fairly systematic series of checks:

```
class Task {
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // check memory address
        if (obj == null || obj.getClass() != this.getClass()) return false;
        Task otherTask = (Task) obj; // downcast after ensuring safety
        if (this.serialNumber == otherTask.serialNumber) {
            return true;
        } else {
            return false;
        }
    }
}
```

- Do not use instanceof here because it returns whether an object is that class or subclass of Type; not as strict and doesn't fulfill symmetry requirement of this method.

`.hashCode()`

Manual Overriding: 1. Select a prime number such as 17. Multiply

2.6 Elements of OOP

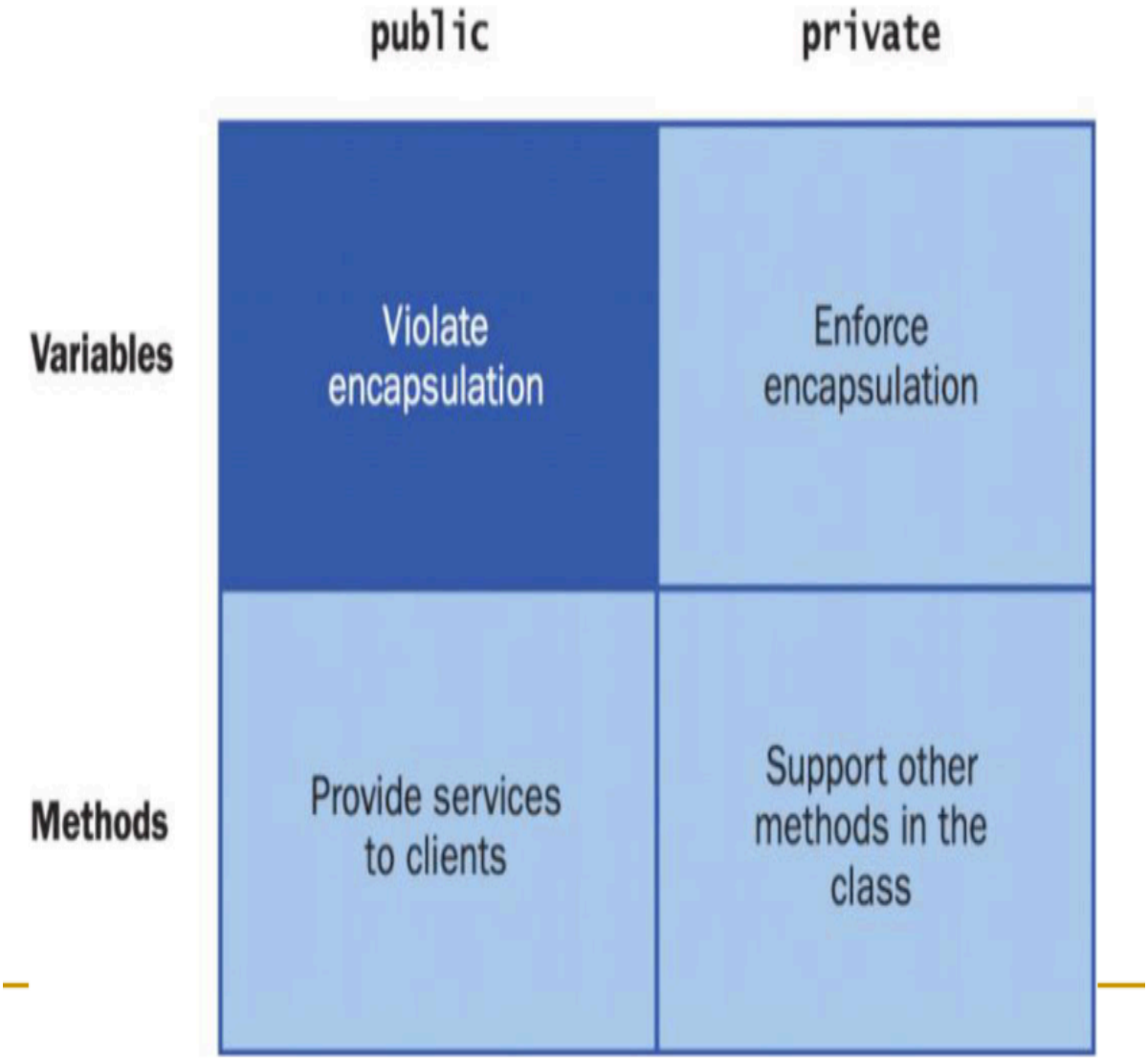
2.6.1 Encapsulation

Encapsulation is the practice of bundling data safely. By not allowing users to arbitrarily change the state of an object, you prevent compatibility and other errors. It is inherently linked to using access modifiers to make code safer against accidental modifications.

2.6.1.1 Best Practices

Fields (instance variables) should typically be private and require getter/setter methods to access, just so that unintended state values are not introduced.

Methods should be public when they are meant to provide a service to the user directly, and private when they are meant to serve as helpers to other internal logic.



2.6.1.2 Access Modifiers By default, classes in Java are package-private if no access modifier is used. This means that the class will only be accessible by other classes within that same Java package.

There are three types of access modifers: - `private` - used for tracking internal state of the class; attributes + helper methods - `protected` - implementation details that need to be accessible to subclasses for inheritance, but not the general public; attributes + helper methods used in subclasses - `public` - well-defined interface that other parts of the program should use to interact with the object; getters + setters + core functionalities

A `private` member, which could be a variable, method, or constructor, can only be accessed from within the class itself.

A `package-private` member (default access level) only allows others within that same package to access the member. It is not available to subclasses unless they are also part of the same package.

A `protected` modifier means that member can be accessed within the same package by any other class. It may also be accessed from subclasses that extend from the parent class containing this protected member.

A `public` member can be accessed from any other class, even if that class is in a different file or package, as long as the package which it belongs to is imported into that other file.

	default	public	private	protected
Same Class	YES	YES	YES	YES
Same package subclass	YES	YES	NO	YES
Same package non-subclass	YES	YES	NO	YES
Different package subclass	NO	YES	NO	YES
Different package non-subclass	NO	YES	NO	NO

2.6.2 Abstraction

Abstraction is the hiding away of complex implementation details.

A class or interface in Java is said to define a contract for the objects which extend/implement them. Classes, or objects, which concretely implement these contracts, define the actual implementation details. However, the precise logic of the code is hidden from the outside world, which is called abstraction.

The only thing that outside users should know about the class is information about its public API (e.g., public methods) such as what a method does, but not necessarily the details of how said method is implemented. This is just like how you need to know how to drive a car, but not how that car actually runs.

The primary benefit of this is that is logic changes must be made, the public API can remain largely unchanged (e.g., method names remain the same), while the internal logic may be shifted.

2.6.2.1 Best Practices

For example, method names should be informative on what they do and largely self-documenting. For example, `getSize()` may return the size of whatever object it's called on, and `isAlive()` basically hints that it returns a boolean value regarding the state of whatever "living/dead" object it's called on.

2.6.3 Inheritance

Inheritance is when a new class, called a subclass, shares some of its fields and methods with a superclass.

The primary benefit of inheritance is code reusability when creating multiple similar-but-different objects, such as creating multiple species of animals. There may be a few things unique about each animal, such as the food that it eats or the sound that it makes, but the overarching theme is the same (e.g., they will all have an `eat()` method and a `makeSound()` method) and have information such as their height and weight.

2.6.3.1 Details Regarding Inheritance in Java

Java only allows for single inheritance, which means that one class can only inherit from a single other class. However, chaining (multi-level inheritance) is allowed (e.g. Animal -> Dog -> GermanShepard).

```
class Dog extends Animal {
    // code goes here
}
```

After inheritance, not only are you able to access instance fields/methods: - `this(xyz)` constructor - `this.xyz` field - `this.xys()` method

You also get access to the parent class's fields and methods (as long as they are not default/private): - `super(xyz)` constructor - `super.xyz` field - `super.xyz()` method

2.6.3.2 The Role of Access Modifiers

A subclass has access to only the public/protected data and methods of the parent, and can call them without specifying that the method comes from the parent class. For example, if `Dog` had the method `.bark()`, then `myGermanShepard` object can directly call `myGermanShepard.bark()`.

2.6.3.3 **Super** and Constructor Chaining

It is a strict rule that `this()` or `super()` must be the very first statement inside a constructor if they are used. A constructor cannot contain both. If no explicit call to `super()` is made, the Java compiler will automatically insert a call to the parent's no-argument constructor. If the parent class lacks a no-argument constructor, a compile-time error will occur.

If a class does not have a constructor, a default zero-argument constructor is automatically provided with a call to the parent class's constructor, which is a `super()` call with no parameters.

Best Practices for Constructor Chaining - When a class has multiple constructors, those with fewer arguments should call the ones with more arguments using `this(...)`. - The constructor with the most arguments should be the one to perform the primary initialization work, including the call to `super()`.

Copy Constructors A copy constructor is a constructor that only takes in a single argument, which is another object of the same class. The constructor's purpose is to copy the values of member variables from an existing object to a new object.

A copy constructor may create either a shallow copy or a deep copy. A shallow copy will have entirely new values for primitive types, but any fields which are references will still point to the same object in memory.

A deep copy creates entirely new objects in memory for all fields.

2.6.3.4 Method Overriding

If a child class defines a function with the exact same function signature as the parent class (with an access modifier that is at least as permissive as the parent), then the method is overriden. - Not to be confused with method overloading. - Can add an `@Override` before the function signature, which doesn't perform the overriding but is a compile-time safety check.

Rules: - The method must have the same name and the same parameter list (number, order, and types of parameters) as the method in the parent class. - The return type must be the same or a subtype (known as a covariant return type) of the

parent method's return type. - The access modifier cannot be more restrictive than the one in the parent class (e.g., a `public` method cannot be overridden as `protected`). - The overriding method cannot throw checked exceptions that are broader or newer than the exceptions thrown by the parent method.

2.6.3.5 Field Shadowing

Field shadowing occurs when a local variable (including function parameters) has the exact same name as an instance field (class attribute).

In constructors, this is okay because the `this` keyword is used to assign argument values to the corresponding field.

However, the following code does nothing because the formal parameter will take precedence. `this` is primarily used to differentiate field shadowing, but it is good practice in general to prefix fields with this when referring to instance fields.

```
class Car {
    private String model; // Instance Field

    public Car(String model) {
        model = model; // Self-assignment
    }
}
```

2.6.4 Polymorphism

Polymorphism is the principle that a single object can take on many forms. For example, a single name (such as `makeNoise()`) can perform different actions when objects are different classes (e.g., `Cow`, `Duck`). - The primary benefit of this is the ability to group related but different objects, such as having an `Animal` array that holds `Dog`, `Cat`, and `Fish` objects. This provides generality in implementation while allowing for specificity in execution, as calling an overridden method on each object will invoke its specific version.

In practice, polymorphism is implemented through inheritance, method overloading, and method overriding.

2.6.4.1 Method Overloading

Method overloading is when a method shares the same name but performs different actions based on a different parameter list (in terms of type or length, or both). This is static polymorphism.

2.6.4.2 Method Overriding

Method overriding is when a child class implements a more specialized version of the same exact method (same name, return type, and parameter list) in the parent class. This is dynamic polymorphism, since the specific implementation is determined at runtime based on the type of the object being referenced. - The `@Override` marker annotation is a sign for the compiler to check whether the method implemented in child class truly matches the signature of parent class's method.

2.6.4.3 Type Casting (Not Primitive Conversion)

Casting is the process of converting a variable from one type to another. In the context of objects, there are three main types of casting.

- Upcasting: This is casting a subclass reference to a superclass reference. It is always safe and can be done implicitly or explicitly.
- Upcasting "masks" any members (fields or methods) that are unique to the subclass, making only the members of the superclass accessible through that reference. It does not change the object's actual dynamic type.

```
Dog myDog = new Dog();
Animal myAnimal = (Animal) myDog; // Explicit upcasting
Animal myAnimal2 = myDog; // Implicit upcasting
```

- Downcasting: This is casting a superclass reference back to a subclass reference. Downcasting always compiles but carries the risk of a `ClassCastException` at runtime.
- This error occurs if the object's actual dynamic type is not compatible with the type it is being cast to. For a downcast to succeed, the object must have been an instance of that specific subclass *(or its descendant)* originally.

```java
// Superclass
class Animal {
    void makeSound() {
        System.out.println("Animal sound");
    }
}

// Subclasses
class Dog extends Animal {
    void fetch() {
        System.out.println("Dog is fetching.");
    }
}

class Cat extends Animal {}

public class DowncastingExample {

    public static void main(String[] args) {

        // --- Safe Downcasting ---
        Animal myAnimal = new Dog(); // Upcasting
        Dog myDog = (Dog) myAnimal; // Downcasting
        myDog.fetch(); // Outputs: Dog is fetching.

        // --- Unsuccessful Downcasting (causes an error) ---
        Animal anotherAnimal = new Cat();
        Dog anotherDog = (Dog) anotherAnimal; // exception here

        // This line is never even reached
        anotherDog.fetch();
    }
}
```

- Side Casting: This refers to casting between two sibling classes (e.g., two classes that share the same parent). Side casting will never compile.

2.6.4.4 Static and Dynamic Types

```java
List<String> fruits = new ArrayList<String>();
```

The static type of a variable is on the left side and declared at compile-time. All methods that are called on that variable must be compatible with this declared static type. - It's good practice to be as loose as possible with the static type. If the actual object can do XYZ, declare the static type to be a class or interface that does XYZ, instead of just XY or Z.

The dynamic type is the actual runtime type of the variable, and determines *which* method is actually used when called at runtime (e.g., if it has been overriden). - For example, a variable can have a static type of a superclass but point to an object with a dynamic type of a subclass: `Person p = new Student();` - Calling `p.getRole()` would call the `Student` class's implementation, not `Person`.

3 Exam 3

Chapters in Book: 13, 20/23/24 ## Abstract Classes and Interfaces ## Abstract Classes An abstract class in Java is declared via: `public abstract class ClassName`. It defines an “is-a” relationship, so that any class that extends it must contain its structure and state. - An abstract class cannot be instantiated on its own, despite having a normal constructor. Instead, the constructor’s only purpose is to be called through `super()`. - An abstract class can have either abstract or concrete methods. However, all abstract methods must be overriden by subclass methods. - Just like with normal class inheritance, a subclass can only extend one abstract class.

```
// Abstract class defines the "is-a" relationship
public abstract class Animal {
    private String name; // Can have state

    public Animal(String name) { // Can have a constructor
        this.name = name;
    }

    public void eat() { // Concrete method
        System.out.println(name + " is eating.");
    }

    public abstract void makeSound(); // Abstract method (the contract)
}

// Subclass must implement the abstract method
public class Dog extends Animal {
    public Dog(String name) {
        super(name); // Call to superclass constructor
    }

    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

3.1 Interfaces

An interface is a purely abstract type that defines a set of common operations or behaviors for objects. It guarantees that any class implementing it will have certain methods available. As such, interface names are often suffixed with “-able” (e.g., `Comparable`). - Everything within an interface is assumed to be `public` and `abstract` (and so such keywords will be automatically inserted regardless of whether they are used). - Cannot contain instance variables or constructors. Only has `public static final` fields. - Must specify value at declaration, as it is final. - One class can implement multiple interfaces.

```
// Interface defines the "can-do" relationship
public interface Trainable {
    void performTrick(); // Abstract method by default
}

// A class can extend another class AND implement an interface
public class Cat extends Animal implements Trainable {
    public Cat(String name) {
        super(name);
    }

    @Override
    public void makeSound() {
        System.out.println("Meow!");
```

```
    }

    @Override
    public void performTrick() {
        System.out.println("The cat is chasing a laser pointer.");
    }
}
```

3.2 The Comparable Interface

The `Comparable` interface specifies that any class which implements it has elements which can be compared to each other and therefore sorted by some natural order.

It has a single public abstract method called `compareTo`, which must return some integer.

Method contract: - Precondition: caller must provide non-null object `o` of specified compatible type (instantiated generic type). - Postcondition: negative int indicates smaller than other object; zero indicates equality; positive indicates greater than.

```
public abstract class Ninja implements Comparable<Ninja> {
    // code here

    @Override
    public int compareTo(Ninja o) {
        // some code here
    }
}
```

3.3 Generic Types

Generic types are used to primarily with collections avoid `ClassCastException` and to ensure compile-time type safety.

Before generic types were introduced, any collection implementing the `List` interface could accept objects of different types, but resulted in issues during accessing or should require explicit casting checks.

```
// Prior
ArrayList dates = new ArrayList(); // no generics
dates.add(new Date());
dates.add(new String()); // doesn't result in compile-time error
Date date = (Date) dates.get(0); // error: explicit casting required

// Now
List<Date> dates = new ArrayList<>();
dates.add(new Date());
dates.add(new String()); // compile-time error
Date date = dates.get(0); // no error
```

Conventional generic types are denoted by a single uppercase letter: - `<E>`: element - `<T>`: type - `<K>`: key - `<V>`: value

They typically appear in angled brackets, during definition of a generic class/interface + during generic instantiation where that generic type is replaced with a concrete type.

```
// "T" is the generic type for whatever goes inside the Box container
public class Box<T> {
    // code here
}

Box<Integer> intBox = new Box<>();
```

3.3.1 Defining Generic Classes and Interfaces

When declaring a class as generic (e.g., `class Box<T>`), then that *entire class* gets that type parameter `T`.

For any non-static method and field, `T` can be treated throughout the class just like any other type, such as `int` or `String`.

```
import java.util.ArrayList;

public class GenericStack<E> {
    private ArrayList<E> list = new ArrayList<>(10);

    // Inserts no-args constructor by default
    // Note: constructor should be defined normally
    // and should not include diamond operator

    public int getSize() {
        return list.size();
    }

    // We don't know the type before this GenericStack is created
    // Allows for parametrized types
    public E peek() {
        return list.get(getSize() - 1);
    }

    public void push(E o) {
        return list.add(o);
    }

    public E pop() {
        E o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    @Override
    public String toString() {
        return "stack: " + list.toString();
    }
}
```

```
// All fields are implicitly public static final
// All methods are implicitly public abstract
public interface Sortable<T extends Comparable<T>> {
    List<T> getSortedList();
    List<T> getSortedList(Comparator<T> comparator);
}
```

- `Comparable<T>` also has `<T>` because the `compareTo` method needs to know the type that it's comparing against.
- This is called a bounded generic type, such that the type passed in must also implement this other interface.

3.3.1.1 Generic Static Methods

A static method *cannot* use the class's type parameter directly, since static methods are at the class level, not an instance. Therefore, generic instantiation does not apply to that static method.

Instead, they declare their own type parameters.

```java
class Box<T> {
    public static <U> void printContents(U[] array) {
        for (U element : array) System.out.println(element);
    }
}
```

- `U` has nothing to do with `T`.
- If instead we said `public static <T> void printContents(T[] array)` then the method's `T` shadows the class's `T` at *method scope*.
  - Within the static method, the type parameter only refers to the method's declaration.
  - Elsewhere, the class's type parameter is used.

3.3.1.2 Bounded Type Parameters

```java
public class StudentData<S extends Comparable<S> & Computable, T> {
    // code here
}
```

- There is no `implements` with generics. Only `extends` is used in method and class headers.
  - `super` may be used with wildcard generics to incidate lower bound, but still not allowed in bounded type parameters.
- When one generic needs to satisfy multiple interfaces, they are joined by the boolean `&` operator.
- If an interface requires a type parameter (most common are `Comparable<T>` and `Iterable<T>`), then the same parametrized generic type must be passed in as well.

Note the distinction between bounded type parameters and wildcard generics. Bounded type parameters are used in the headers of generic methods and classes, where the type parameter is named (e.g., `T`) and can be referenced throughout the code logic. Wildcard generics refer to some unknown subtype/supertype (denoted by `?`) and cannot be referenced but allows for safe + flexible APIs with containers.

3.3.2 Generic Instantiation

```java
GenericStack<Integer> stack1 = new GenericStack<Integer>();
GenericStack<Integer> stack2 = new GenericStack<>(); // infer type
```

3.3.3 Generic Methods

This is an alternative to generic classes if the class doesn't need to store data of a generic type (i.e., no generic fields). This is common for utility methods, where you pass in something to get an output rather than relying on the state of a specific instance.

The generic type must be placed immediately before the return type of the method, which will allow Java to know that we aren't specifying a class of name T, but rather a placeholder type.

```java
import java.util.List;
import java.util.ArrayList;

public class Utils {
    public static <T> T getFirstElement(List<T> list) {
        if (list == null || list.isEmpty()) {
            return null;
        }
        return list.get(0); // the return type is T
    }
}
```

3.3.4 Covariance in Arrays + Motivation for Wildcard Generics

When initializing an array in Java the dynamic type of the array is able to be a subtype of the static type.

```
Animal[] animals = new Dog[2];
```

This allows some incorrect code to compile but throw an error at runlike, such as `ArrayStoreException`: `animals[1] = new Cat()`.

However, generics are invariant. So even if `A` is a subclass of `B`, `List<A>` is not a subclass of `List<B>`. That means that a line of code such as `List<Animal> animals = new ArrayList<Dog>()` will NOT compile. Furthermore, you cannot pass `List<Dog>` to a method expecting `List<Animal>`.

3.3.5 Wildcard Generics and Covariance

The common problem is that with a class like `Animal` and subclasses like `Cat` and `Dog`, they are treated entirely differently despite an inheritance relationship in terms of Generic type checking.

For example, although a `Dog` is an `Animal`, a `List<Dog>` is not a `List<Animal>` for type safety (e.g., if both `Dog` and `Cat` objects may be added to an array `List<Animal> animalList` , then there will be exceptions at runtime when trying to retrieve objects).

With the introduction of upper-bounded wildcards, `List<A>` is able to become a subtype of `List<? extends B>` (covariance).

However, if we only wanted to read from the list, then the operation is safe. Here is an example with a method in a generic class

```
// Upper-bounded; don't know the exact subclass but knows that they are all Animals
// Good for reading elements (guaranteed to get an Animal)
public void printAnimals(List<? extends Animal> list) {
    for (Animal a : list) {
        System.out.println(a);
    }
}

// Lower-bounded; works for all parent classes
// Good for writing (any Dog/above element works) but bad for reading since you don't know which type
public void addDog(List<? super Dog> list) {
    list.add(new Dog()); // maybe the list contains heterogeneous types
}
```

- The upper/lower both include the type itself. E.g., `Animal` / `Dog` are included respectively.

This leads to the PECS principle: - Producers Extend: When the `extends` keyword is used, we are upper-bounded and we know that the code is meant to product output (via reading). - Consumers Super: When the `super` keyword is used, we are lower-bounded and we know that the code is meant to consume input (and use it via writing).

3.3.6 Bounds: Producer Extends, Consumer Super

The syntax `<? extends Type>` means that the wildcard represents some type that extends or implements `Type`, making `Type` the upper bound.

In this case, the collection can only as a “read-only producer”, allowing for the accessing of elements. This is safe because when reading elements as `Type`, it’s always safe to upcast (that is, have the static type be less specific than the dynamic type).

The syntax `<? super Type>` means that the wildcard represents some type that is `Type` or is a superclass of `Type`.

In this case, the collection can only as a "consumer" (gaining elements). Then adding elements of `Type` is safe because all `Type` upholds the "is-a" relationship for any superclass of `Type`. - For example, adding `Student` to an array of unknown supertype (e.g., `List<Person>`) is safe because all `Student` objects can be treated as a `Person` object. - Anything trying to read/get from the collection results in a compile-time error, except for `Object o = collection.get(0)`. - This is because everything is guaranteed to be an `Object`, so this is safe. Otherwise, the compiler cannot assume anything.

Compilation Rules: - The right side (dynamic type) must fit within the bounds defined on the left side (static type). - Left side defines allowed behavior (e.g., what methods are supported). - Right side determines the actual logic that is implemented once the method is called at runtime.

3.3.7 Type Erasure During Compilation

Generic type information does not exist at runtime due to type erasure during compilation. This means that type compatibility checks occur at compile-time, and if the code properly compiles, then the generic type is: 1. Replaced by `Object` if the type was unbounded. 2. Replaced by specified bound if the generic type was bounded. 3. Type casts are inserted at points of retrieval.

```
// Pre-compilation
List<String> list = new ArrayList<>();
String s = list.get(0);

// After type erasure
List<Object> list = new ArrayList<>();
String s = (String) list.get(0);; // inserted typecast
```

3.4 Reading/Writing Data

3.5 Exceptions

3.5.1 Error Hierarchy

The exception hierarchy in Java is rooted in the `Throwable` class, which has two main branches: `Exception` and `Error`.

The `Exception` class represents conditions that a reasonable application might want to catch and handle, while the `Error` class represents serious problems that applications should not attempt to catch, as they typically stem from the environment rather than the code itself.

Under the `Exception` class, there is a special subclass called `RuntimeException`. This subclass divides exceptions into two categories: checked exceptions (all `Exception` subclasses that do not inherit from `RuntimeException`) and unchecked exceptions (all subclasses of `RuntimeException`). - Additionally, all subclasses of `Error` are considered unchecked.

3.5.2 Checked Exceptions

Checked exceptions are exceptions that the compiler requires you to handle within your code. These exceptions represent recoverable conditions that can reasonably be anticipated and handled by the application. All subclasses of `Exception` that do not inherit from `RuntimeException` are checked exceptions.

- Common examples include `IOException` and `ClassNotFoundException`
- The compiler enforces that these exceptions must either be caught using a try-catch block or declared in the method signature using the `throws` keyword

Notably: - All checked exceptions must be handled within the try-catch block or conveyed in the method header (to pass responsibility to caller). - If the caller doesn't fulfill responsibility, code does not compile. - If the try-catch block is too specific in the types of exceptions that it catches (e.g., some unchecked exceptions are not explicitly caught), then the code still doesn't compile.

Conventions: - Have the more specific exceptions caught higher up in the try-catch-except block, since they are executed in order. You want the most specific exception message to be communicated and not masked by a different broader exception.

3.5.3 Unchecked Exceptions

Unchecked exceptions are exceptions that the compiler does not require you to explicitly handle. These exceptions typically indicate programming errors or conditions that are difficult to recover from. All subclasses of `RuntimeException` and `Error` fall into this category.

- Common examples include `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `IllegalArgumentException` (which occurs when the right type is passed but with a wrong value)
- Since these are unchecked, methods are not required to declare them in their signature, and callers are not forced to catch them
- Runtime exceptions generally indicate bugs in the code that should be fixed rather than conditions that need to be handled at runtime

3.5.4 Methods of Handling Exceptions

3.5.5 Throw an Exception

To purposefully throw (raise) an exception, use `throw new SpecificException("Message")`.

3.5.5.1 Method Header

This defers the responsibility of using a try-catch-finally block to its caller (who can then choose to defer upward further).

```
// If multiple, can be separated by commass
public void processFile(String filename) throws FileNotFoundException, CustomException {
    // code here
}
```

3.5.5.2 Try-Catch-Finally

```
try {
    // code that may throw multiple exceptions
} catch (FileNotFoundException e) {
    // Most specific
} catch (IOException e) {
    // Less specific
} catch (Exception e) {
    // Least specific
} finally {
    // This code always executes
}
```

- If there is a more general exception below a more specific exception (e.g., a superclass), then that code will be considered unreachable and Java will not compile it.

3.6 File IO

Generally, there are two types of files: text (.txt, .csv) and binary (.class, .exe).

To work with either kind, it's necessary to import the `java.io` package, or use `Scanner` from `java.util`.

3.6.1 Buffering and Disk Writing

When performing write operations to a file, it doesn't immediately go to the disk. Instead, it is temporarily stored in a buffer/memory to reduce direct disk access.

Once the buffer is "full", disk writing occurs. This naturally leads to the question of what happens if the buffer is not full but data must still be written to the disk.

"Flushing" is when data in the buffer is written to disk. Flushing occurs either when `.flush()` on the stream/writer is called or when `.close()` is implicitly/explicitly called. - If the program crashes while there is still stuff in the buffer, that is simply lost data.

3.6.2 Checked Exception: IOException

`IOException` is a superclass for lots of other exceptions that might occur, like `FileNotFoundException`.

It is also a checked exception, so it needs to be handled in the code.

```
import java.io.FileReader;
import java.io.IOException;

// Including throws in the header passes responsibility of handling to the caller
// Caller must either have this in try-catch or pass responsibility up further
public void processFile(String filename) throws IOException {
    FileReader reader = new FileReader(filename);
    try {
        // code
    } finally {
        reader.close() // always good practice to close
        // if the program crashes, it is not implicitly closed
    }
}

// OR
public void processFile(String filename) {
    FileReader reader = new FileReader(filename);
    try {
        // code
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (reader != null) {
            try {
                reader.close(); // may also throw exception
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

3.7 Data Structures and Objects

3.8 Abstract Data Types

An Abstract Data Type (ADT) is a high-level programming concept which focuses entirely on "what" a data structure is able to do. It defines the conceptual model.

An *interface* is the purest realization of an ADT, focusing only on defining the contract and required operations. - Defines "can-do" relationship.

Alternatively, an *abstract base class* is another way of realizing an ADT, but through a common base definition and partial implementation of logic. - Defines "is-a" relationship.

3.9 List Interface

A `List` is an ordered collection, also called a sequence. Lists allow for the insertion of elements at arbitrary indices and allow for duplicates.

Informational Methods: - `int indexOf()`: return first index of object - `int size()`: return number of elements in the list - `boolean isEmpty()`: return whether list is empty (zero elements) - `boolean contains()`: return whether list contains specific element - `boolean containsAll()` –boolean equals()‘: return whether this list is equivalent to another list; must be same size and contain all the same elements in the same ordering

Operational Methods: - `boolean add()`: append element to end of list - `boolean addAll()` - `boolean remove()`: remove first occurrence of element in list; if not present, returns false and leaves list unchanged - `boolean removeAll()` - `void clear()`: remove all elements from list - `default void sort()`: sorts array given all elements implement the `Comparable` interface

Index Modification Methods: - `E get(int index)`: return element at provided integer index - `E set(int index, E element)`: replaces element at specified index with new element; returns old element - `void add(int index, E element)`: add element at index; shifts all elements (including element previously occupying that position) to the right - `void remove(int index)`: remove element at index and returns it; shifts subsequent elements to the left

3.9.1 ArrayList

An ArrayList is a concrete class which can be imported via `java.util.ArrayList`. It only stores reference objects (no primitive types) and has a dynamic size. - It is called an ArrayList because it implements the `List` interface and uses an array behind the scenes. - Though it cannot store primitive types, it can store their wrapper classes, so this limitation is hardly limiting in rpactice.

```
import java.util.ArrayList;

// Creates new ArrayList with initial capacity of 20
// Still empty when first created (unlike built-in arrays which start with null)
// Primary benefit of specifying initial capacity is to reduce resizing computational costs
ArrayList<String> list = new ArrayList<String>(20);

// Infer the type + use default initial capacity
ArrayList<string> list = new ArrayList<>();
```

- Upon initialization, the arraylist is empty (`.size() == 0`), not null.

3.9.2 LinkedList

3.10 Iterators and Iterable

3.10.1 Iterable Interface

Found in `java.lang` package.

An `Iterable<T>` interface provides two key methods: - `.iterator()`: returns an iterator over elements of type `T` - `.forEach(Consumer <? super T>)`: performs given action for each element of Iterable until all elements have been processed

`.iterator()` is called under the hood when using an enhanced for loop. It is first called to return an `Iterator` object, and then continuously invokes the `hasNext()` and `next()` methods to access each element.

3.10.2 Iterator Interface

Found in `java.util` package.

An `Iterator<E>` interface provides the following methods: - `.hasNext()`: returns whether iteration has more elements - `.next()`: returns next element - `.remove()`: removes last element returned by iterator from the underlying collection

3.11 Algorithms

3.12 Searching

3.12.1 Linear and Binary Search

```
public class Search {

    public static int linearSearch(int[] numbers, int target) {
        for (int i = 0; i < numbers.length; i++) {
            if (numbers[i] == target) {
                return i;
            }

            return -1; // not found
        }
    }

    public static int binarySearch(int[] sortedNumbers, int target) {
        int left = 0;
        int right = sortedNumbers.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            int x = sortedNumbers[mid];
            if (x == target) {
                return x;
            } else if (x < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return -1; // not found
    }
}
```

3.13 Sorting

3.13.1 Quadratic Time

```
// All O(n^2) time
public class QuadraticSort {

    // - Selects the smallest (or largest if sorting in reverse) and swaps
    // wth numbers[i-1] on ith iteration, maintaining an unsorted tail
    // Skips last element (naturally largest since we chose min each time)
    // - Not stable sorting due to swapping mechanism
    public static void selectionSort(int[] numbers) {
        // i is the element that we move minElement to
        for (int i = 0; i < numbers.length - 1; i++) { // last element is sorted
            int minIdx = i;
            for (int j = i + 1; j < numbers.length; j++) {
                if (numbers[j] < numbers[minIdx]) {
                    minIdx = j;
                }
            }
            if (minIdx != i) { // if a new smallest idx is found
                int temp = numbers[i];
```

```
                numbers[i] = numbers[minIdx];
                numbers[minIdx] = temp;
            }
        }
    }

    // - Starts at 2nd element; first is naturally relatively min in sorted region
    // - Inserts key element into proper locatin within sorted head
    // - If key is not in correct position, first shifts all displaced elements
    // one position rightward
    // - Performs the swap last, accounting for one-off indexing
    public static void insertionSort(int[] numbers) {
        for (int i = 1; i < numbers.length; i++) {
            int key = numbers[i]; // stores "temp"; allows for replacement/shifting
            int j = i - 1;
            while (j >= 0 && numbers[j] > key) {
                numbers[j + 1] = numbers[j];
                j--;
            }
            numbers[j + 1] = key;
        }
    }

    // - Start from the beginning with pairwise swaps to bubble max element to
    // a sorted right tail
    // - Inner for loop's range is determined by boundary of sorted region
    // - Have a boolean flag to check whether there can be an early exit
    public static void bubbleSort(int[] numbers) {
        for (int i = 0; i < numbers.length - 1; i++) {
            boolean swapped = false;
            for (int j = 0; j < numbers.length - i - 1; j++) {
                if (numbers[j] > numbers[j + 1]) {
                    int temp = numbers[i];
                    numbers[i] = numbers[i + 1];
                    numbers[i + 1] = temp;

                    swapped = true;
                }
            }

            if (!swapped) {
                return;
            }
        }
    }
}
```

3.13.2 LogLinear Time

```
class LogLinearSort{
    // Recursive, in-place merge-sort algorithm
    // Inclusive right bounds for both mergeSort() and merge()
    // Overload mergeSort() function for public API vs. private logic
    public static void mergeSort(int[] numbers) {
        // Temp array to store values during merge step w/o overwriting elements
        int[] temp = new int[numbers.length];
        mergeSort(numbers, temp, 0, numbers.length - 1);
    }
```

```
    private static void mergeSort(int[] numbers, int[] temp, int left, int right) {
        if (right - left == 0) {
            return; // only a single-element interval is sorted by default
        }

        int mid = left + (right - left) / 2;
        mergeSort(numbers, temp, left, mid);
        mergeSort(numbers, temp, mid + 1, right);
        merge(numbers, temp, left, right);
    }

    private static void merge(int[] numbers, int[] temp, int left, int right) {
        int mid = left + (right - left) / 2;

        // Make copies of left/right to avoid modifying original bounds
        int i = left;
        int j = mid + 1;
        int k = left; // the actual location of pointer at element to be inserted

        // The contents of temp doesn't really matter; it all gets copied over
        // in one "atomic" step at the end of this function to the main array
        while (i <= mid && j <= right) {
            if (numbers[i] <= numbers[j]) {
                temp[k++] = numbers[i++];
            } else {
                temp[k++] = numbers[j++];
            }
        }
        while (i <= mid) {
            temp[k++] = numbers[i++];
        }
        while(j <= right) {
            temp[k++] = numbers[j++];
        }
        for (int idx = left; idx <= right; idx++) {
            numbers[idx] = temp[idx];
        }
    }
}
```

4 Final Exam

4.1 Collection Interfaces

4.2 The Collection Interface

4.2.1 Definition and Properties

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method	Description
boolean	add(E e)	Ensures that this collection contains the specified element (optional operation).
boolean	addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection to this collection (optional operation).
void	clear()	Removes all of the elements from this collection (optional operation).
boolean	contains(Object o)	Returns true if this collection contains the specified element.
boolean	containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
boolean	equals(Object o)	Compares the specified object with this collection for equality.
int	hashCode()	Returns the hash code value for this collection.
boolean	isEmpty()	Returns true if this collection contains no elements.
Iterator<E>	iterator()	Returns an iterator over the elements in this collection.
default Stream<E>	parallelStream()	Returns a possibly parallel Stream with this collection as its source.
boolean	remove(Object o)	Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
boolean	retainAll(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	size()	Returns the number of elements in this collection.
default Spliterator<E>	spliterator()	Creates a Spliterator over the elements in this collection.
default Stream<E>	stream()	Returns a sequential Stream with this collection as its source.
Object[]	toArray()	Returns an array containing all of the elements in this collection.
default <T> T[]	toArray(IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.
<T> T[]	toArray(T[] a)	Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

Note that methods marked as optional are not truly optional in that you cannot opt out of implementing them as if they had a default implementation. Instead, it means that when overriding and providing a specific implementation for the interface, you may opt to throw an `UnsupportedOperationException`.

4.3 Stack

4.3.1 Manual Implementation

The hierarchy for the built-in stack object goes: `Object -> AbstractCollection -> AbstractList -> Vector -> Stack`. The following is a simple array-backed stack class written from scratch.

```
public class ArrayStack<T> {
```

```
    T[] data;
    int size;

    public ArrayStack() {
        this(10);
    }

    @SuppressWarnings("unchecked")
    public ArrayStack(int initialCapacity) {
        if (initialCapacity <= 0) {
            throw new IllegalArgumentException();
        }
        this.data = (T[]) new Object[initialCapacity];
        this.size = 0;
    }

    public void push(T item) {
        if (this.size > this.data.length) {
            this.expandCapacity();
        }
        this.data[this.size++] = item;
    }

    public T pop() {
        if (this.isEmpty()) {
            throw new java.util.EmptyStackException();
        }
        T item = this.data[--this.size];
        this.data[this.size] = null;
        return item;
    }

    public T peek() {
        if (this.isEmpty()) {
            throw new java.util.EmptyStackException();
        }
        return this.data[this.size - 1];
    }

    public boolean isEmpty() {
        return this.size == 0;
    }

    public int size() {
        return this.size;
    }

    @SuppressWarnings("unchecked")
    private void expandCapacity() {
        T[] temp = (T[]) new Object[this.data.length * 2];
        for (int i = 0; i < this.size; i++) {
            temp[i] = this.data[i];
        }
        this.data = temp;
    }
}
```

4.4 ADT: Set

4.4.1 Definition and Properties

A set needs to contain unique elements in an unordered manner. When adding an element to a HashSet (most common implementation), Java first calculates the object's hash code using its `hashCode()` method to know where to store it, and then checks for strict equality using `equals()`. Only if both match then the element is considered a duplicate and rejected.

Unlike Python, which requires sets store immutable (hashable) objects, Java allows for the storage of mutable collections like ArrayLists by default (technically any Java object can be stored, as `equals()` and `hashCode()` are inherited). - However, this is not suggested, because if modifications are made via the original reference to that mutable container, its hashcode changes (as hashcode is calculated from internal contents) but in the Set its "bucket" has not changed.

All Methods | **Instance Methods** | **Abstract Methods** | **Default Methods**

Modifier and Type	Method	Description
boolean	`add(E e)`	Adds the specified element to this set if it is not already present (optional operation).
boolean	`addAll(Collection<? extends E> c)`	Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	`clear()`	Removes all of the elements from this set (optional operation).
boolean	`contains(Object o)`	Returns `true` if this set contains the specified element.
boolean	`containsAll(Collection<?> c)`	Returns `true` if this set contains all of the elements of the specified collection.
boolean	`equals(Object o)`	Compares the specified object with this set for equality.
int	`hashCode()`	Returns the hash code value for this set.
boolean	`isEmpty()`	Returns `true` if this set contains no elements.
`Iterator<E>`	`iterator()`	Returns an iterator over the elements in this set.
boolean	`remove(Object o)`	Removes the specified element from this set if it is present (optional operation).
boolean	`removeAll(Collection<?> c)`	Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	`retainAll(Collection<?> c)`	Retains only the elements in this set that are contained in the specified collection (optional operation).
int	`size()`	Returns the number of elements in this set (its cardinality).
default `Spliterator<E>`	`spliterator()`	Creates a `Spliterator` over the elements in this set.
`Object[]`	`toArray()`	Returns an array containing all of the elements in this set.
`<T> T[]`	`toArray(T[] a)`	Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

4.4.2 Implementation

```
import java.util.Set;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class ArraySet<E> implements Set<E> {
```

```
    private E[] elements;
    private int size;

    public ArraySet() {
        this(10);
    }

    public ArraySet(int initialCapacity) {
        this.elements = (E[]) new Object[initialCapacity];
        this.size = 0
    }

    @Override
    public boolean isEmpty() {
        return this.size == 0;
    }

    @Override
    public boolean add(E element) {
        if (element == null) {
            throw new NullPointerException();
        }

        if (this.contains(element)) {
            return false;
        }

        if (this.size == this.elements.length) {
            this.expandCapacity();
        }

        this.elements[size++] = element;
        return true;
    }

    @Override
    public boolean contains(Object element) {
        for (int i = 0; i < this.size; i++) {
            if (this.elements[i].equals(element)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public boolean remove(Object element) {
        for (int i = 0; i < this.size; i++) {
            if (this.elements[i].equals(element)) {
                this.elements[i] = this.elements[--this.size];
                this.elements[this.size] = null;
                return true;
            }
        }
        return false;
    }

    @Override
```

```
    public Iterator<E> iterator() {
        return new ArraySetIterator(); // returns user-defined object
    }

    private class ArraySetIterator implements Iterator<E> {
        int index = 0; // index of the object to return next

        @Override
        public boolean hasNext() {
            return this.index < size;
        }

        @Override
        public E next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            return elements[index++];
        }
    }

    private void expandCapacity() {
        E[] temp = (E[]) new Object[this.elements.length * 2];
        for (int i = 0; i < this.size; i++) {
            temp[i] = this.elements[i];
        }
        this.elements = temp;
    }
}
```

For any object that implements `Iterable`, the enhanced `for` loop is turned during compilation into a normal `for` loop that calls `iterator()` and then uses `hasNext()` and `next()` methods to move forward:

```
for (Element e : someIterable) {
    // some code
}

// BECOMES

for (Iterator<Element> it = someIterable.iterator(); it.hasNext(); ) {
    Element e = it.next();
    // some code
}
```

4.5 ADT: Map, HashMap

4.6 ADT: Set, HashSet

4.7 Spring Boot

Spring Boot is a framework used for building backends, such as internal tools and REST APIs, that simplifies the development process compared to plain Java web applications. While a standard Java web app requires manual configuration of a servlet container like Tomcat and extensive XML configuration files, Spring Boot automatically handles this boilerplate setup. It utilizes the Spring framework underneath to manage dependencies and configurations, allowing developers to focus on business logic rather than infrastructure.

4.8 Project Architecture

The architecture of a Spring Boot application follows a structured flow starting from the client and ending at the database. The browser sends HTTP requests which are intercepted by the Controller, which acts as a router handling the incoming request paths. The Controller delegates complex operations to the Service layer, which contains the core business logic. For data persistence, the Service layer communicates with the Repository, which utilizes Spring Data JPA to interact with the Database. Finally, a View layer may be used to render HTML templates to the user, although many modern applications simply return JSON.

The execution of a Spring Boot application begins with a class annotated with `@SpringBootApplication`. This annotation triggers the startup sequence, which includes initializing the embedded configuration (such as starting the Tomcat server & JPA database connection) and scanning the project for components to instantiate as Spring Beans. The standard Java `main` method serves as the trigger, calling `SpringApplication.run` with the primary class and command-line arguments.

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.SpringApplication;

@SpringBootApplication // autoconfiguration
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}
```

4.9 Beans and Inversion of Control

Spring manages application components through a concept called Inversion of Control (IoC), where the framework controls object creation rather than the developer.

This results in us not needing to use the `new` keyword. Any class such as Controllers, Services, and Repositories are automatically instantiated and managed as Beans (essentially singleton instances automatically inserted by the program wherever necessary). Developers define these dependencies in the class constructor, and Spring injects the necessary instances at runtime.

4.9.1 Stereotype Annotations

Various annotations identify the role of each class to the Spring container: - `@Component`: Designates a general class managed by Spring. - `@Service`: Marks a class containing business logic. - `@Repository`: Marks a class responsible for database access. - `@Controller`: Indicates a web controller that returns views (HTML). - `@RestController`: A specialized controller that returns data directly, typically as JSON.

4.9.2 Dependency Injection

Dependency injection basically gets rid of the need for developers to instantiate new instances of Service, Repository, etc. Instead, just include the `@Autowired` annotation before a constructor, and then you're set (just define the constructor + class fields the same way as usual).

One primary benefit of dependency injection is that it makes things simpler when testing the application, decoupling the instantiation of dependencies (other objects) from the logic we're trying to test.

```
package com.example.demo.service;

import org.springframework.stereotype.Service;

@Service
public class UserService {

    public String getUserGreeting(String username) {
```

```
        return "Hello, " + username + "! Welcome from the UserService.";
    }
}
```

```
package com.example.demo.controller;

import com.example.demo.service.UserService;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class UserController {

    // 1. Declare the dependency as a final field
    private final UserService userService;

    // 2. Use the constructor for injection
    // The @Autowired annotation is optional for a single constructor
    // in Spring Boot 2.3 and later, but included here for clarity.
    // @Autowired
    public UserController(UserService userService) {
        this.userService = userService; // 3. The dependency is injected here
    }

    @GetMapping("/greet")
    @ResponseBody
    public String greetUser(@RequestParam String name) {
        // 4. Use the injected dependency
        return userService.getUserGreeting(name);
    }
}
```

4.10 MVC Architecture

The Model-View-Controller (MVC) pattern dictates the flow of data: Browser → Controller → Model (+ Repository) → View. When building a feature, it is generally best to follow a specific implementation order to ensure dependencies are available when needed. The recommended sequence is to first define the Model, followed by the Repository, then the Service, the Controller, and finally the View.

4.11 Model

The Model represents a basic unit of data or single entry in the database. These classes are marked with `@Entity` to indicate they map to a database table. The primary key of the table is denoted by `@Id`. Fields can be automatically populated using `@GeneratedValue`, often with a strategy like `GenerationType.AUTO`, which provides default values such as auto-incrementing IDs.

Models are just normal Java classes, marked with `@Entity`.

```
package com.example.demo.model;

import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
```

```java
// 1. Model Class Definition

@Entity
public class User {

    // 2. Fields (Properties)

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
    private String email;

    // 3. Constructors (REQUIRED for frameworks)

    // Default No-Argument Constructor (REQUIRED by JPA and Jackson)
    public User() {
    }

    // Constructor with all fields (useful for data creation)
    public User(String firstName, String lastName, String email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }

    // Constructor including the ID (useful for reading data)
    public User(Long id, String firstName, String lastName, String email) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }

    // 4. Getters and Setters (REQUIRED for Jackson binding and data access)

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
```

```
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

4.12 Repository

The Repository layer handles data access and typically consists of an interface that extends `JpaRepository<T, ID>`, where `T` is the model type and `ID` is the primary key type. This inheritance provides built-in methods for standard database operations without requiring custom SQL. - Standard methods include `save(entity)`, `findAll()`, `findById(id)`, `deleteById(id)`, `count()`, and `existsById(id)`.

Additional "custom" (but also provided-by-default) query methods are included if method signatures are specified within the interface, using the guideline of: - `<action>By<field>` (i.e., `findByEmail(String email)`)

```
package com.example.demo.repository;

import com.example.demo.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import java.util.List;
import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    Optional<User> findByEmail(String email);

    List<User> findByLastNameStartingWith(String prefix);

    List<User> findByFirstNameOrLastName(String firstName, String lastName);
}
```

Notice that the repository is an interface. By Inversion of Control, Spring sees that it's a `@Repository` and extends `JpaRepository<T, ID>` and uses a factory to create a class/proxy to implement all methods defined in JpaRepository.

4.13 Service

The Service layer consists of classes annotated with `@Service` that contain the actual backend Java code. These classes deal exclusively with Java objects and business logic. Since they are managed beans, their instantiation is handled by Spring Boot, and they are typically injected into controllers via the constructor.

Similar to models, service classes are basically normal Java code, and deal with Java objects (no HTTP or anything), focusing purely on backend implementation logic. Services often interact with the respository, as it is necessary to transform data.

```
package com.example.demo.service;

import com.example.demo.model.User;
import com.example.demo.repository.UserRepository;
import org.springframework.stereotype.Service;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import java.util.List;
import java.util.Optional;

@Service
public class UserService {

    // 1. Dependency Declaration: The UserRepository interface
    private final UserRepository userRepository;

    // 2. Constructor Injection: Spring injects the generated implementation
    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // 3. Business Logic Methods

    /** Saves a user. Can be used for both creation and update. */
    public User saveUser(User user) {
        // Here you might add validation logic before saving
        return userRepository.save(user);
    }

    /** Retrieves all users. */
    public List<User> findAllUsers() {
        return userRepository.findAll();
    }

    /** Retrieves a user by ID. */
    public Optional<User> findUserById(Long id) {
        return userRepository.findById(id);
    }

    /** Retrieves a user by email, using the custom method defined in the repository. */
    public Optional<User> findUserByEmail(String email) {
        // Logic might include checking if email is valid before calling the repository
        return userRepository.findByEmail(email);
    }

    /** Deletes a user by ID. */
    public void deleteUser(Long id) {
        userRepository.deleteById(id);
    }
}
```

4.14 Controller

The Controller defines the endpoints for the application. A class can be assigned a base URL path using `@RequestMapping("path/to/service")`. Methods within the controller are mapped to specific HTTP actions using annotations like `@GetMapping` or `@PostMapping`. To handle dynamic data, controllers use `@PathVariable` to extract values from the URL path and `@RequestBody` to map the HTTP request body to a Java object.

There are two types of controller annotations: @Controller and @RestController.

Feature	`@Controller`	`@RestController`
Primary Use	Web Pages (Server-Side Rendering)	REST APIs (JSON/XML)
Return Value	View Name (String) *Example:* `"signup"` → `signup.html`	Data Payload (JSON) *Example:* `{"user": "alice"}`
Mechanism	Uses a ViewResolver to find and render a template.	Uses MessageConverters (like Jackson) to write directly to the response body.
Context	Used when serving HTML directly to a browser.	Used when serving data to a frontend (React, Mobile App, etc.).

4.14.1 Mappings

A class can be assigned a base URL path using `@RequestMapping(/api/service)`. This is essentially prefixed every time. For example, for a specific resource such as "call", we could have different endpoints like "call/list" and "call/update". Instead of having to write this every time, using `@RequestMapping("/call")` essentially adds the "/call" prefix to everything.

There are a few different types of HTTP Requests: - GET - retrieves data from a specified resource (e.g., loading a webpage, fetching a list of items) - POST - submits data to the specified resource, often causing a change in state on the server side - PUT - replaces a specific representation of a resource with a new one - PATCH - basically PUT but with partial modifications instead of complete replacement - DELETE - deletes the specified resource

For each type of request, we have an annotation named `@<Req-Type>Mapping("path/to/endpoint")`. For example, `GetMapping("/list")` or `PostMapping("/user")`.

By convention, the endpoint paths are nouns, while the type of mapping (i.e., the HTTP request type) is a verb dictating the nature of the action.

Note: Sometimes, things will not need a further endpoint specification beyond what is defined in `@RequestMapping`. In that case, just use `@<Req-Type>Mapping` without any parentheses / endpoing string as the parameter.

4.14.2 Extracting Data from Endpoint Path

`@PathVariable` is used in front of parameter names in order to extract path variables for use. - The name of the path variable must match the name of the parameter.

```
@GetMapping("/users/{userId}/orders/{orderId}")
public Order getSpecificOrder(
    @PathVariable Long userId,
    @PathVariable Long orderId
) {
    // code here
}
```

4.14.3 Extracting Data from Requests

`@RequestBody` is used to bind the entire body of the HTTP request to a method parameter. - Delegates JSON deserialization to Jackson library converter. - Compares the keys in the incoming JSON object with field names in target Java class (in this case, `User`). - Uses all-arguments constructor or setters defined in the model to construct the object.

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User newUser) {
    // 'newUser' is a fully populated User Java object,
    // deserialized from the JSON sent in the request body.
    User savedUser = userService.save(newUser);
    return new ResponseEntity<>(savedUser, HttpStatus.CREATED);
}
```

4.14.4 Preparing Data for the View

Java Object –> HTML Template `modelInstance.addAttribute("attributeName", new modelClass())` can be used so that a blank new output can be shown upon a `GET` request for the user's browser.

Model (Regular Java Object)

```
package com.example.demo.model;

public class SignupForm {
    private String username;
    private String email;

    // Constructors
    public SignupForm() {} // Empty constructor needed for binding

    // Getters and Setters (Crucial for Spring to bind data!)
    public String getUsername() { return username; }
    public void setUsername(String username) { this.username = username; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

Controller

```
package com.example.demo.controller;

import com.example.demo.model.SignupForm;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

@Controller
public class SignupController {

    // 1. GET Request: Prepare the "Blank Sheet of Paper"
    @GetMapping("/signup")
    public String showSignupForm(Model model) {
        // Create an empty object so the HTML form has something to bind to
        model.addAttribute("signupForm", new SignupForm()); // name provided should match th:object

        return "signup"; // name of the HTML file to render (signup.html)
    }

    // 2. POST Request: Receive the "Filled Sheet"
    @PostMapping("/signup")
    public String processSignup(@ModelAttribute SignupForm signupForm) {
        // Spring has already filled 'signupForm' with the user's input!

        return "redirect:/success"; // redirect to a different page
    }
}
```

HTML (ThymeLeaf)

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
```

```
<head>
    <title>Sign Up</title>
</head>
<body>
    <h1>Create an Account</h1>

    <!--
      th:object="${signupForm}" -> Binds the form to the object we added in the GET controller
      th:action="@{/signup}"    -> Submits data to the POST /signup endpoint
    -->
    <form th:action="@{/signup}" th:object="${signupForm}" method="post">

        <label>Username:</label>
        <!-- th:field="*{username}" -> Binds this input to signupForm.setUsername() -->
        <input type="text" th:field="*{username}" />
        <br/><br/>

        <label>Email:</label>
        <!-- th:field="*{email}" -> Binds this input to signupForm.setEmail() -->
        <input type="text" th:field="*{email}" />
        <br/><br/>

        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

`@ModelAttribute` is used to bind request parameters (from a form or query string) to a Java object.

On a Method Parameter (Receiving Data) * Usage: `public String process(@ModelAttribute SignupForm form)` * Action: Spring looks at the incoming request data (e.g., `username=alice&email=test@test.com`) and automatically fills the fields of your `SignupForm` object. * Matching Logic: It matches Parameter Names to Class Setters (e.g., `username` -> `setUsername()`). * Request Body: It parses the standard form data (key-value pairs) from the request body or URL parameters.

Flow 1. GET Request (**`showSignupForm`**): * Controller creates an empty `SignupForm` object. * Adds it to the `Model` (`addAttribute`). * Returns `"signup"` (View Name). * Result: User sees an empty form. 2. User Types Data & Submits: * Browser sends a POST request with data: `username=bob&email=bob@gmail.com`. 3. POST Request (**`processSignup`**): * Spring sees `@ModelAttribute SignupForm form`. * Spring creates a new `SignupForm` instance. * Spring calls `form.setUsername("bob")` and `form.setEmail("bob@gmail.com")`. * Controller receives the filled `form` object ready to use.