Q8.1

Java’s Generics uses type erasure. The concept involves the compiler to pre-process the code and type check it before compilation and erase all of that information from bytecode during compilation. In order for this to work i.e. in order for the information to be wiped out, the type we place in <> brackets has to be replaced with Object type (the compiler also inserts casts where needed). Primitives are not objects, therefore, we cannot have a LinkedList<int> but we can have a LinkedList<Integer>.

new T() is created at run time but java doesn’t know what T was anymore since it has been erased during compilation.

Q8.2

Please see the repository for the code.

Q8.3

A wildcard is represented by a question mark ? and it means “any type”. So if we have List<?> list it means list is a list of any type. We can put fancy bounds on the wildcard. There is a lower bound and an upper bound. A lower bound involves a key word extends.

e.g.

List<? extends Number> eg1 = new ArrayList<Number> //is ok

List<? extends Number> eg2 = new ArrayList<Double> //is ok

List<? extends Number> eg3 = new ArrayList<Integer> //is ok

List<? extends Number> eg4 = new ArrayList<Object> //is not ok because Object does not extend Number

So for a lower bound we can use anything that is “below” Number hierarchically. We can read from the list because we know that anything would be of type Number but we cannot write anything to the list because the compiler doesn’t actually know what type it is dealing with.

An upper bound involves a key word super.

e.g.

List<? super Double> eg5 = new ArrayList<Double> //is ok

List<? super Double> eg6 = new ArrayList<Object> //is also ok

We can write type double and can read type Object.

Q8.4

(a) Looking at the addItems method, both lists are of type T so both lists have to be of the same type. Since generics in java are invariant List<Double> and List<Number> are not of the same type even though Double extends Number. (Could we go through covariance vs invariance so that I am sure I understand please?)
Hence the code in main does not compile.

(b) Since we read from src then the list should be of type <? extends T> and since we write to dest then the list should be of type <? super T>. Please see the repository for implementation.

Q8.5

Please see the repo

Q9.1

Please see the code in repo

Q9.2

Could we please go over checked vs unchecked exception?

I asked ChatGPT about the difference between those and just wanted to share with you how hilarious its answer was, full of analogies, so that you can have a good laugh too. And the explanation actually makes sense!

Start of chatGPT answer:

Checked vs unchecked exceptions are basically **two different personality types** in Java's "problem reporting" system.

🎭 The Two Types of Exceptions

Think of Java exceptions as two kinds of coworkers:

✔ Checked Exceptions — "The Formal, Rule-Following Coworker"

These are the coworkers who insist:

“If something *might* go wrong, you *must* plan for it.
I won't let you ignore me.”

Example: IOException, SQLException, FileNotFoundException

Java forces you to:

- either **handle** them with try/catch,
- or **declare** them with throws.

If you don’t, your code simply will not compile.

Analogy:

It’s like a coworker who says:

“Look, this might blow up. I need you to *sign this form* that says you’re prepared.”

These exceptions represent **recoverable problems** — things your program could reasonably handle:

- file missing
- network down
- database temporarily unavailable

These are “expected” problems in the real world.

⚠ Unchecked Exceptions — “The Wild, Don’t-Warn-You Coworker”

These guys are like:

“If I cause trouble… you’ll find out when it happens. Good luck.”

Example: NullPointerException, IllegalArgumentException, ArithmeticException

These extend RuntimeException.

Java does *not* force you to handle them.

You can try/catch them, but you don’t have to.

Analogy:

This coworker never hands you paperwork.
They just walk in and suddenly say:

“The server is on fire. Surprise!”

Unchecked exceptions usually represent **bugs**:

- You dereferenced a null pointer
- You gave a method an invalid argument
- Your logic went sideways

You *should fix the code*, not “handle” these exceptions most of the time.

End of chatGPT answer

> (a,b) We should use checked exception here because it's a real-world problem and not just bad code that we should fix i.e. the input is a user input and not a bug made by a developer. Therefore, it must be handled.

Q9.3

Please see repo for code

Q9.4

We have a separate exception for when the power is 0. If the power n is 0 then we return 1. Otherwise, we just recursively multiply x until we hit base case and power n = 0. I feel like it was unnecessary to make an exception in this particular case since power n = 0 is just a base case in our recursion. Also I think the code breaks for negative powers, and this is where we should have used an exception instead.

Q9.5

I am trying to figure this question out but I feel like I lack enough java experience to be able to answer this correctly. I know that java doesn't use destructors because of garbage collection. Garbage collection monitors the moment when the object is not referenced anymore and frees up memory. I am not sure how to implement the process when it comes to destructors though. Do we just set every field of the object to 0? But that still occupies memory. Or do we use null instead??

Ah I checked out the lecture slides and it says that with resources we use the .close() method. Or have I misunderstood something?

Q9.6

Finally block is always executed. So 6 is not returned immediately. If finally doesn't interfere then 6 is returned. If finally also returns some different number then that number is our output instead of 6. It's like finally block is always superior then the try block and is always the priority.

Q10.1

State pattern allows an object to cleanly alter its behaviour when its internal state changes.

Strategy pattern allows us to cleanly interchange between different algorithm implementations.

Q10.2

(a) We have a problem that arises when a lecture gets promoted to a professor. If we do not use state pattern we will have to delete a lecturer and replace it with a professor. But it might not be possible to delete it everywhere in the program. If it's in the payroll then the person could be paid twice, as a lecturer and as a professor.
(b) We must not let anyone or anything else have access to mRank because then the problem that a lecturer can be lost somewhere in the program and not replaced by the professor arises again.
(c) Do we create a new class that is used for getting a rank? Could we go over this please?

Q10.3

(a) The draw method does not necessarily have to be abstract because we use an interface. The interface already implies that the implementing classes have to define their own draw methods (I think. Right?)
(b) Please see repo
(c) Please see repo
(d) Please see repo

Q10.4

Please see repo for code. I think I got a bit lost in my own code but the idea was to create 2 interfaces, a wrappable book and a book. Then the wrappedBook class can only accept and work with wrappable books…

Q11.1 and 11.2 Please see the repo

EXAM QUESTIONS

2022 Q4

(a) Type checking happens at compile time. The compiler checks whether the types that a developer used are consistent everywhere. For example, if we have a list of type Integer, the compiler checks whether the elemets added to the list are of type Integer. This is type checking. It prevents the program from crashing.

(b) Type erasure happens during compilation. The compiler "forgets" all the type checking it's done before compilation. For example, we cannot write new T() because the compiler already forgot what T is, everything becomes an Object in bytecode. Java uses type erasure to be compatible with its previous versions.

(c) (i) True because Integer extends Number.
(ii) False because Generics are invariant in java.
(iii) True because the wildcard represents any single type that extends Number.
(iv) Now there is no restriction on the wildcard so I think this one is false. The mark scheme says it's true and I don't get why.
(v) I think this is similar to the previous one and there is also no restriction on the wildcard and hence false.

(d) new TreeSet () implies that it's of type Object. New TreeSet<Object> () explicitly tells you that it's of type object. When we use the latter, our code becomes more readable for you and for your team (if applicable) especially when sets of other types are involved. New TreeSet<>() is used only when TreeSet<Object> ts = new TreeSet<>() because there is no need to repeat the type again.

2025 Q4

(a) A class should be closed to modification but open for extension.

(b) Inheritance, Polymorphism, Encapsulation, Abstraction.
Encapsulation hides the data and makes it not accessible from outside of the class i.e. closed to accidental modification.
Inheritance involves children mimicking the behaviour of its parent. The fields of the parent can be accessed inside the class itself and inside its subclasses. Inheritance is used for design patterns like decorators and strategy patterns which can be used for implementing extensions.

Polymorphism involves every class overriding a method to suit their own criteria. Abstraction and polymorphism can be used to make new componenets to inherit from base classes allowing new behaviour.

(c) (i) The code will compile. We can read from the list, and the types seem to be consistent.
(ii) It will not compile because we cannot write anything to the numList. NumList is a list of some type that extends Number but the compiler does not know what type exactly.
(iii) We can only write Integers into intList. 6 is an int which is then boxed into an Integer (I think) and hence the code will compile.
(iv) intList can be a list of any type that is “above” Integer hierarchically, e.g. Number, Object. So the code will compile.
(v) The code will not compile. We can only read Object from the list. We don’t know for sure what type we are dealing with so we cannot assume that i is an Integer. We could write Object i = .. which would be okay I think.

2022 Q3

(a) Fields of immutable classes cannot be accessed from outside of the class. The behaviour of the class cannot be changed. If you are working in a big team then it’s very important that the class cannot be accidentally changed by one of your teammates or by another class for that matter. Immutable class makes it easier to debug. However, creating an object every time means more memory usage.
(b) Please see the repo
(c) Could we go through this part please?