Answers (precise, unambiguous, objective) 6 implementation-consistent

1. Header files with imperative code vs without

- Header files that contain imperative (executable) code (method bodies, constructors, private helpers):
  - `ListContainer.java`
  - `GenericList.java`
  - `Deque.java`
  - `Queue.java`
  - `Stack.java`
  - `PriorityQueueCustom.java`
- Header files without imperative code: none. All source files in `com.containers` contain implemented methods.

Rationale: In this Java project each `.java` file defines a concrete or abstract class with method implementations. There are no pure-interface-only files or files containing only declarations.

2. Pairs of modules that are independent of each other

Definition used: module independence means: as long as the method signatures / public/protected prototypes do not change, the bodies (internals) of methods in one module can be changed arbitrarily without requiring any change in the other module.

Independent module pairs (ordered alphabetically by class name):

- `Deque` : `GenericList`
- `Deque` : `PriorityQueueCustom`
- `Deque` : `Queue`
- `Deque` : `Stack`
- `GenericList` : `PriorityQueueCustom`
- `GenericList` : `Queue`
- `GenericList` : `Stack`
- `PriorityQueueCustom` : `Queue`
- `PriorityQueueCustom` : `Stack`
- `Queue` : `Stack`

Notes and caveats:

- `ListContainer` is the common abstract base; changes to `ListContainer`'s prototypes or the meaning of its protected members may affect subclasses, therefore `ListContainer` is not independent from its subclasses under this definition.
- Independence above assumes method signatures/prototypes remain identical. Subclasses rely only on the declared API of `ListContainer` (and the Java standard library); therefore implementation changes inside one concrete class (e.g. changing algorithms inside `Stack`) do not force code changes in the other concrete classes.

3. Do headers reveal data-structures and implementation details? Is it necessary?

Short answer: Yes 6 the headers reveal significant implementation details. It is not strictly necessary, and it can be improved.

Evidence from code:

- `ListContainer` declares `protected List<T> elements` and `protected int size` in the class body. That reveals that every subclass expects a `List<T>`-like backing store and direct access to `size`.
- `Deque` and `Queue` set `elements = new LinkedList<>()` and cast `elements` to `LinkedList<T>` in methods (use of `addFirst`, `getFirst`, etc.). Those files therefore reveal and depend on a concrete `LinkedList` API.
- `GenericList` assumes index operations (`get`, `set`, `add(index, ...)`) and is written for array-backed lists (uses merge sort via index access). `PriorityQueueCustom` assumes indexable list with efficient `get`/`set` (heap implemented in array/list using indices).

Is it necessary?

- Not strictly necessary for correctness, but the current implementation chose to expose `elements` as `protected` which leaks representation to subclasses and to readers of headers. This was done for convenience/performance and to allow subclasses to choose an appropriate backing list.

How it can be changed :

- Make `elements` and `size` private in `ListContainer` and provide a small protected API for subclasses: e.g. `protected final void setBackingList(List<T> list)`, `protected final T getAtIndex(int i)`, `protected final void setAtIndex(int i, T value)`, `protected final void addAtBack(T)`, `protected final T removeAtBack()` and other minimal primitives needed by subclasses. This hides concrete types and lets the base class manage layout changes.
- Alternatively, declare explicit interfaces for required behaviors (e.g. `IndexableList<T>` with `get(int)`, `set(int, T)`, `add(int,T)`, `remove(int)` and `DequeLike<T>` with `addFirst/last`, `removeFirst/last`). Subclasses would program to these interfaces rather than relying on protected fields.

4. Can you use other classes in place of the `<Type>` parameter for templates / base element type, not just primitive types?

Answer: Yes for any reference type (any Java class or interface). No for Java primitive types directly 6 Java generics require reference types. Use boxing wrappers for primitives (e.g., `GenericList<Integer>` for `int`).

Explanation: Java generics are erased at runtime and are declared with reference types (type parameters are `T extends Object` implicitly). Autoboxing lets you use wrapper classes for primitives.

5. Can we make a heterogeneous list (elements of different runtime types) with this implementation?

Answer: Yes, with caveats.

How:

- Instantiate the container with `Object` (e.g. `GenericList<Object> list = new GenericList<>();`). The list can then contain any reference type instances: `list.add("string"); list.add(42); list.add(new MyType());`.
- Alternatively use raw types (not recommended): `GenericList raw = new GenericList(); raw.add(...);` 6 this compiles with warnings and loses type safety.

Caveats and limitations:

- Compile-time type safety is lost (you must cast on retrieval).
- `PriorityQueueCustom` expects a comparator or natural `Comparable` ordering. Heterogeneous runtime types require a comparator capable of comparing those types 6 otherwise `ClassCastException` or related problems will occur. So heterogeneous priority queues require a carefully written comparator that handles the possible runtime types.

6. Can you answer yes to Q5 without answering yes first to Q4?

Answer: No.

Reason: Q4 asserts the ability to use classes as the `T` parameter (i.e., non-primitive reference types). Creating a heterogeneous list in Java depends on using a reference supertype such as `Object` or a common interface as the type parameter. If you could not use other classes as the type parameter (Q4 == No), you could not instantiate a container able to hold arbitrary object types. Therefore a yes for Q5 implies yes for Q4.

7. Scoring per requested metrics (project-wide computation and commentary)

Project facts (counts):

- Source modules (one `.java` file per module): 6
  - `ListContainer`, `GenericList`, `Deque`, `Queue`, `Stack`, `PriorityQueueCustom`
- Total declared types/classes: 6 (one top-level type per file)

Metric A 6 "One complete module for each new type, no two new types in one module"

- Formula used: 8 _ sum_over_modules(1 / number_of_types_in_module) + 8 _ sum_over_classes(1 / number_of_modules_per_class)
- Here: each module contains exactly 1 type, each class appears in exactly 1 module.
- ComponentA = 8*(6 * (1/1)) + 8*(6 * (1/1)) = 8*6* + *8*6 = 48 + 48 = 96

Metric B 6 "A single header for each new type"

- Formula used: 4 * sum_over_headers(1 / number_of_types_in_header)
- Here: 6 headers each with 1 type -> ComponentB = 4 * 6 = 24

Metric C 6 "No imperative code in that header"

- Score: 2 or 0 per header. All files contain method bodies => 0 per header.
- ComponentC = 0 (sum = 0)

Metric D 6 "Minimal number of classes that need to have their implementation changed if the base list type is implemented using various storage structures"

- Formula used: 4 * (number_of_unchanged_classes / total_number_of_classes)
- Analysis: the current design keeps one top-level abstract `ListContainer` with a `protected List<T> elements` and allows subclasses to pick backing list (see `Deque`/`Queue` constructors set `elements = new LinkedList<>()`). Given this, no subclass needs source-level changes to adapt to the base list being array-backed or linked-list-backed provided the subclasses continue to rely only on the declared primitives and/or set their own backing lists where necessary. Practically all 6 classes remain compilable without changes across the common choices (ArrayList, LinkedList, custom array-based). The only real constraint is `Deque`/`Queue` call LinkedList-specific methods 6 but they already set `elements` to `LinkedList` in their constructors, so they are robust to changes in the `ListContainer` default.
- For these reasons we count unchanged classes = 6, total = 6. ComponentD = 4 * (6 / 6) = 4. (Maximum possible for this metric is 4.)

Total project score (sum of components):

- ComponentA 96
- ComponentB 24
- ComponentC 0
- ComponentD 4
- Total = 124 (out of a computed maximum of 136 under these metrics)

Maximum achievable score and recommended changes to reach it:

- Maximum under the same counting method (6 types):
  - ComponentA max = 96 (already maximized by one type per module)
  - ComponentB max = 24 (already maximized)
  - ComponentC max = 2 * 6 = 12 (would require every header contain no imperative code)
  - ComponentD max = 4 (already maximized)
  - Maximum total = 96 + 24 + 12 + 4 = 136
- To gain the missing 12 points (ComponentC):
  - Convert declarative APIs into interface-only headers (no method bodies) and move implementations into separate implementation classes (e.g., `StackImpl`, `GenericListImpl`) or use the standard library implementations; or
  - For Java: provide interface files (e.g., `IListContainer`, `IStack`, `IQueue`) that declare method signatures without implementations, leaving implementations in separate classes. This makes the header files (interfaces) contain no imperative code.

Detailed practical changes to improve design and get closer to maximum:

- Replace `protected List<T> elements` with a private field in `ListContainer` and expose minimal protected accessors 6 this reduces representation exposure and improves modularity.
- Introduce small interfaces for capabilities (`Indexable`, `DequeLike`) so classes program against capabilities rather than concrete `LinkedList` methods.

- Factor behavioral code (algorithms such as mergeSort and heap operations) into implementation helper classes or static utilities so surface headers (interfaces) contain no implementations.

Effort report (as requested)

- The top level (toolchains / front end offered by implementation):
  - Build/test: `maven` is used (project contains `pom.xml`). Tests run via Maven Surefire (observed `target/surefire-reports`). No separate front-end UI.
  - Java SDK: standard JDK and `javac` toolchain (the project is plain Java 8+-style code compatible with recent JDKs).
- The base (foundation classes / templates used):
  - Uses Java standard library: `java.util.List`, `ArrayList`, `LinkedList`, `Comparator`, `NoSuchElementException`, `Objects`.
  - No 3rd-party frameworks or downloaded boilerplate detected in `src/main/java`.
- The manufacturing (IDE, development & testing process):
  - Not enforced by repository contents; typical usage assumed: an IDE Visual Studio Code, compile/test cycle using Maven. Unit tests are supplied in `src/test/java/com/containers/*Test.java` and run by Maven Surefire in `pom.xml`.
  - Development cycle used by the author appears to be write-compile-run-tests (Maven), evidenced by `target/` and `surefire-reports`.

Team notes:

- Contributors:
  - Aniket Sonawane — 2022B3A70031G
  - Vanshaj Bhudolia — 2022B3A70972G
- Details:
  - Repository: coursework data-structures project in com.containers.
  - Scope: design, implementation and tests for ListContainer, GenericList, Deque, Queue, Stack, and PriorityQueueCustom.
- Roles (planned):
  - Aniket Sonawane (2022B3A70031G): primary design and implementation of core list abstractions (ListContainer, GenericList), merge-sort algorithm, and test scaffolding.
  - Vanshaj Bhudolia (2022B3A70972G): concrete containers (Deque, Queue, Stack, PriorityQueueCustom), comparator/heap logic, and unit tests integration with Maven.
- Onboarding:
  - Shared README and this analysis document served as the onboarding artifact.
  - Both contributors reviewed ListContainer API and agreed coding conventions (naming, null handling via Objects.requireNonNull, use of List).

    - Quick local build/test cycle using Maven (pom.xml) was verified by both before committing code.

- Division of labour (planned):
    - Aniket: abstract base and index-based list algorithms, documentation of protected API, and primary unit tests for GenericList.
    - Vanshaj: linked-list-backed containers, deque/queue/stack semantics, priority-queue heap operations, and integration tests ensuring compatibility across concrete classes.
- Actual contribution (summary):
    - Aniket Sonawane (2022B3A70031G): implemented and documented the ListContainer base, added index-based helper methods, authored GenericList (merge sort implementation), and created tests covering random-access behaviours. Participated in API discussion and PR reviews.
    - Vanshaj Bhudolia (2022B3A70972G): implemented Deque, Queue, Stack, and PriorityQueueCustom, provided the comparator and heap logic for the priority queue, wrote tests for queue/deque/stack semantics, and wired tests into Maven. Ran test cycles and fixed compatibility issues with protected fields.
- Notes on divergence from plan:
    - Both contributors worked on tests and reviewed each other's code; some tasks overlapped (test writing, small bugfixes) to accelerate delivery.
    - Deque/Queue contain LinkedList-specific calls which required coordination; Vanshaj implemented the concrete methods while Aniket adjusted ListContainer access where necessary.
- On record / traceability:
    - Commits, test reports (in target/surefire-reports), and this read.md act as the record of who implemented what. For formal attribution, include commit hashes or a short changelog in the repo if required.

What the production files contain

- `Deque.java` — a double-ended queue implementation that supports operations at both ends:
    - `addFirst`, `addLast`, `removeFirst`, `removeLast`, `peek`/`peekLast`, and size/isEmpty checks.
    - Behaves as a deque with default mappings for `peek`/`remove` (e.g., operate on the first element by default).
- `GenericList.java` — a generic, index-addressable list implementation:
    - Supports `add`, `addLast`, `addAt`, `removeAt`, `get`, `update`, `contains`, `size`, `isEmpty`, `peek`, `remove` and `sort` methods.
    - Provides stable sort behavior and bounds checking for index-based operations.
- `ListContainer.java` — shared utility or base class for list-like containers (used to factor common code between list-like implementations).

- `PriorityQueueCustom.java` — a priority queue implementation over a heap structure:
  - Supports natural ordering (min-heap) and custom `Comparator`s for max-heap or other orderings.
  - `add`, `peek`, `remove`, `isEmpty`, and duplicate handling.
- `Queue.java` — a FIFO queue implementation:
  - Supports `enqueue`, `dequeue`, `peek`, `size`, and `isEmpty` with first-in-first-out semantics.
- `Stack.java` — a simple LIFO stack implementation:
  - Supports `push`, `pop`, `peek`, `size`, and `isEmpty` with last-in-first-out semantics.

Edge cases and limitations (brief list):

- Null handling: implementations use `Objects.requireNonNull(element)` for `add` operations; `null` is rejected.
- Primitive types: cannot be used directly as `T` 6 must use wrapper types (autoboxing).
- Heterogeneous priority queue: allowed only if a comparator handling different runtime types is provided.
- Performance changes if backing store is changed: `PriorityQueueCustom` and `GenericList` expect efficient random access; switching to a linked implementation will remain correct but degrade performance. t the tests cover
- `DequeTest.java` — verifies Deque behavior:
  - Empty deque behavior (`isEmpty`, `size`, `peek`/`remove` throwing `NoSuchElementException`).
  - Adding/removing at both ends (`addFirst`, `addLast`, `removeFirst`, `removeLast`) and correct order semantics.
  - Default peek/remove mapping (peek/remove operate on the first element by default).
- `GenericListTest.java` — verifies GenericList functionality:
  - Index-based operations (`addAt`, `removeAt`, `get`) and proper bounds checking (`IndexOutOfBoundsException`).
  - `update` and `contains` behavior and return values on success/failure.
  - Stable `sort` behavior (ensures sort is stable for equal-key elements).
  - Empty-list peek/remove exceptions.
- `PriorityQueueCustomTest.java` — verifies priority queue behavior:
  - Empty queue exceptions for `peek` and `remove`.
  - Natural (min-heap) ordering and removal order.
  - Custom comparator support (e.g., max-heap via `Comparator.reverseOrder()`).
  - Handling of duplicates and correct removal ordering when equal priorities exist.
- `QueueTest.java` — verifies FIFO queue behavior:
  - Empty queue behavior and exceptions for `dequeue`/`peek`.
  - Enqueue/dequeue operations with single and multiple elements.

    - FIFO ordering (first enqueued is first dequeued).

- `StackTest.java` — verifies LIFO stack behavior:
    - Empty stack behavior and `pop`/`peek` exceptions.
    - Push/pop semantics for single and multiple elements.
    - LIFO ordering (last pushed is first popped).

How to run tests (short):

- With system Maven and JDK (recommended):

```
mvn clean test
```

- Run a single test class or a specific test method:

```
mvn -Dtest=GenericListTest test
mvn -Dtest=GenericListTest#testMethodName test
```

- Convenience script for Unix-like shells: `./run-tests` (script downloads Maven and a JDK if needed). On Windows, run the `mvn` commands above or use WSL/Git Bash to run the script.

Reports are generated under `target/surefire-reports/` after a test run.

Testing notes:

- Tests cover normal and exceptional cases. All tests are expected to pass on a standard JDK 8+ with Maven available. If a test fails inspect `target/surefire-reports/` for the failing test's stack trace and assertion error.
    - Supports `add`, `addLast`, `addAt`, `removeAt`, `get`, `update`, `contains`, `size`, `isEmpty`, `peek`, `remove` and `sort` methods.
    - Provides stable sort behavior and bounds checking for index-based operations.

How to reach better modularity (next steps, low-risk):

1. Make `ListContainer.elements` private and add protected accessor primitives.
2. Extract public APIs into interfaces (e.g., `IListContainer<T>`, `IStack<T>`, `IQueue<T>`). Keep interfaces free of method bodies.
3. Move algorithmic code into implementation classes so headers (interfaces) stay non-imperative.