Multiple-Choice Quiz — Week 12

Generics and Collections · for reference: Weeks 9–10

Question 1. In Java 7 and later, what does the diamond operator `<>` let you omit when creating a generic object?

- ☐ **A)** The type arguments on the right-hand side of the assignment
- ☐ **B)** The class name on the left-hand side
- ☐ **C)** The parentheses in the constructor call
- ☐ **D)** The angle brackets in the variable's declared type

Question 2. At runtime, does Java still know the actual type used for a generic type parameter like `<T>`?

- ☐ **A)** No — it is erased
- ☐ **B)** Yes — it is preserved
- ☐ **C)** Only for `String` and other built-in types
- ☐ **D)** Only if the class has just one constructor

Question 3. In `List<String> names = new ArrayList<>();`, what does `String` specify?

- ☐ **A)** The type of elements the list holds
- ☐ **B)** The initial size of the list
- ☐ **C)** The name of the list variable
- ☐ **D)** The class that implements the list

Question 4. What does Java do automatically when you write `list.add(5)` on a `List<Integer>`?

- ☐ **A)** It boxes the primitive `int` into an `Integer` object automatically
- ☐ **B)** It throws a `RuntimeException`
- ☐ **C)** It stores the value as a `String`
- ☐ **D)** It silently ignores the call

Question 5. True or false: a `Set` can be implemented using a `List` as its underlying storage.

- ☐ **A)** True
- ☐ **B)** False

Question 6. True or false: a `List` can be implemented using a `Set` as its underlying storage.

- ☐ **A)** True
- ☐ **B)** False

Question 7. Which operation is a `LinkedList` typically faster at than an `ArrayList`?

- ☐ **A)** Getting the element at a random index
- ☐ **B)** Inserting or removing an element at the front of the list
- ☐ **C)** Sorting all of its elements
- ☐ **D)** Checking whether the list is empty

Question 8. A `HashMap<String, Integer> ages` contains no entry for the key `"Sam"`. What does `ages.get("Sam")` return?

- ☐ **A)** `0`
- ☐ **B)** `null`
- ☐ **C)** It throws a `NoSuchElementException`
- ☐ **D)** It fails to compile

Question 9. What is the name of the special method that Java runs automatically when an object is created with `new`?

- ☐ **A)** The constructor
- ☐ **B)** The destructor

□ **C)** The `main` method
□ **D)** The static initializer

Question 10. A class declares one `static` field. After creating three objects of that class, how many separate copies of that field exist?

□ **A)** Three, one per object
□ **B)** One, shared by the whole class
□ **C)** Zero, until an object accesses it
□ **D)** It depends on whether the field is `private`

Question 11. When an overridden method is called through a superclass-typed variable, which version runs?

□ **A)** The subclass's overridden version, via dynamic dispatch
□ **B)** The superclass's version, since that's the declared type
□ **C)** Both versions run, one after another
□ **D)** It fails to compile

Question 12. Which keyword does a class use to declare that it provides implementations for every method listed in an interface?

□ **A)** `extends`
□ **B)** `implements`
□ **C)** `interface`
□ **D)** `abstract`

Answer Key — Week 12 Quiz

Concept tested in *italics*.

Q1 **A** *Diamond operator.* `<>` lets the compiler infer the type arguments from the declared variable type, so they need not be repeated on the right-hand side. The class name, constructor parentheses, and the declared type's own angle brackets are all still required.

Q2 **A** *Type erasure.* Generic type information is erased at compile time, so at runtime the JVM retains no record of what `T` was. This is why you can't do things like `new T[10]` or `T.class` directly — only `Object`-based workarounds are possible.

Q3 **A** *Generic type parameter.* `<String>` fixes the type of elements the list can hold — here, only `Strings`. It does not set a size, name a variable, or specify an implementing class.

Q4 **A** *Autoboxing.* `list.add(5)` automatically boxes the primitive `int` into an `Integer` via `Integer.valueOf(5)`, since a generic collection can only hold objects, not primitives. The call succeeds without throwing, and the value is stored as an `Integer`, not a `String`.

Q5 **A** *Implementing a `Set` with a `List`.* True — a `Set`'s only real requirement is uniqueness. You can implement one by storing elements in a `List` and checking (`contains`) before every `add`, rejecting duplicates. It's less efficient than a hash-based `Set`, but it works.

Q6 **A** *Implementing a `List` with a `Set`.* True — pair each element with its index (e.g. store `(index, value)` entries in the `Set`) and a `Set` can recover ordering, indexed access, and duplicate-tolerance, since every pair is unique even when two values repeat.

Q7 **B** *ArrayList vs. LinkedList.* `LinkedList` needs only to relink a node to add or remove at the front — $\mathcal{O}(1)$ — while `ArrayList` must shift every element down, taking $\mathcal{O}(n)$. `ArrayList` wins at random access, and neither list type sorts faster than the other or has a faster `isEmpty()`.

Q8 **B** *HashMap lookups.* `get` on a missing key returns `null` rather than throwing — there is no equivalent of an "index out of bounds" for maps. `0` would only appear if you explicitly unboxed the result, which would in fact throw a `NullPointerException`, not return `0`.

Q9 **A** *Constructors (Week 9 review).* The constructor runs automatically on `new` to initialize a fresh object's fields. Java has no destructor, `main` is only the program's entry point, and static initializers run once per class, not per object.

Q10 **B** *Static fields are shared (Week 9 review).* A `static` field belongs to the class itself, so there is exactly one copy no matter how many objects exist. Access level (`private` or not) has no bearing on how many copies exist.

Q11 **A** *Polymorphism (Week 10 review).* Java performs dynamic dispatch: the object's actual runtime type determines which overridden version runs, regardless of the variable's declared type. The call compiles fine, runs deterministically, and only one version executes.

Q12 **B** *Interfaces (Week 10 review).* `implements` is used to fulfil an interface's method contract. `extends` is for inheriting from a superclass (or extending another interface), `interface` declares the interface itself, and `abstract` marks a class or method as incomplete.