1Z0-808 Exam Topic Reviewer

TopicId: 1026
Topic: Generics

August 5, 2025

Generics: Writing Type-Safe Code

Alright class, let's discuss Generics. Before Java 5, collections like `ArrayList` were a bit like a wild party—anyone could get in. You could put a `String`, an `Integer`, and a `Cat` object into the same list. This led to chaos at runtime when you tried to pull things out and cast them. Generics impose order. They are Java's way of creating type-safe containers, ensuring you only put the right kind of objects into a collection and eliminating the need for messy casts and runtime errors.

0.1 The Power of Type Safety

Generics add a type parameter to a class or interface, specified in angle brackets `<>`.

```
// Before Generics (dangerous)
List list = new ArrayList();
list.add("text");
list.add(123); // No error here...
String s = (String) list.get(1); // ...but ClassCastException at RUNTIME!

// With Generics (safe)
List<String> stringList = new ArrayList<>(); // Diamond operator <>
stringList.add("text");
// stringList.add(123); // COMPILE ERROR! Problem caught early.
String s2 = stringList.get(0); // No cast needed.
```

Generics turn runtime exceptions into compile-time errors, which are always better.

0.2 Generic Classes and Methods

You can create your own generic classes and methods.

- **Generic Class:** A class that can work with any object type. `T` is a type parameter that will be replaced by a real type when an object is created.

  ```
  public class Crate<T> {
      private T contents;
      public void pack(T contents) { this.contents = contents; }
      public T unpack() { return contents; }
  }
  Crate<Apple> appleCrate = new Crate<>();
  ```

- **Generic Method:** A method with its own type parameter. The type parameter is declared before the return type.

  ```
  public <E> void printElements(E[] elements) {
      for (E element : elements) {
          System.out.println(element);
      }
  }
  ```

0.3 Bounded Wildcards: The Exam's Favorite Trap

This is where you must focus. A wildcard `?` represents an unknown type. Bounded wildcards restrict that unknown type.

- **Upper Bound - `<? extends Type>`:** Represents `Type` or any of its subclasses. Think of it as "read-only." You can retrieve elements (they are guaranteed to be at least of `Type`), but you **cannot add** elements (except `null`) because the compiler doesn't know the exact subtype.

  ```
  // This method can accept List<Number>, List<Integer>, List<Double>, etc.
  public void processNumbers(List<? extends Number> list) {
      for (Number n : list) { /* read is OK */ }
      // list.add(Integer.valueOf(5)); // COMPILE ERROR!
  }
  ```

- **Lower Bound - `<? super Type>`:** Represents `Type` or any of its superclasses. Think of it as "write-only." You **can add** elements of type `Type` or its subclasses, but when you read from it, you can only be sure you're getting an `Object`.

  ```
  // This method can accept List<Integer>, List<Number>, List<Object>
  public void addIntegers(List<? super Integer> list) {
      list.add(Integer.valueOf(5)); // OK
      list.add(10); // OK
      // Integer i = list.get(0); // COMPILE ERROR! Could be a Number or Objec
  }
  ```

Remember the mnemonic **PECS**: **P**roducer **E**xtends, **C**onsumer **S**uper.

Key Takeaways for the 1Z0-808 Exam

- Generics provide compile-time type safety and eliminate the need for casting.
- The diamond operator `<>` simplifies instantiation of generic classes.
- Master bounded wildcards. Use `extends` for read-only access to a hierarchy. Use `super` for write-only access.
- Understand that due to **type erasure**, generic type information is not available at runtime. The compiler handles all the checks.