GUVI | HCL

Skill Up. Level Up

Generics & Lambda Expressions

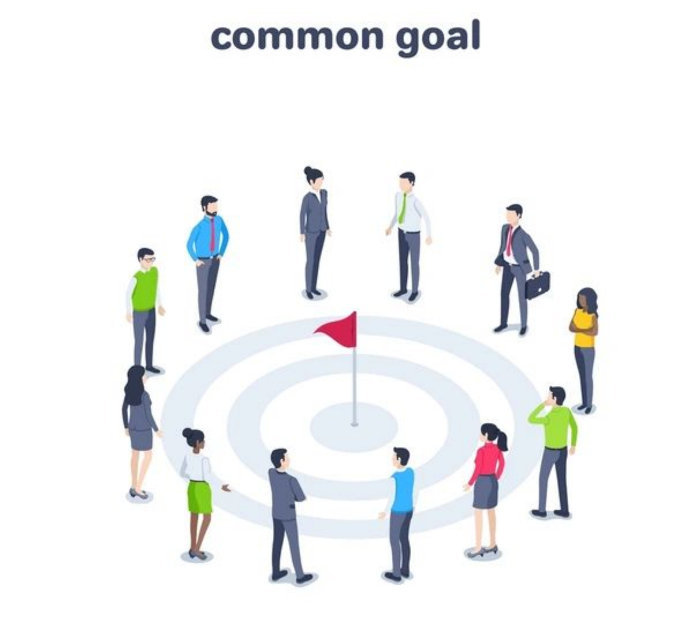
Generics & Lambda Expressions

Presented by
Nikhil Nair

Website
www.guvi.com

Objectives

- What you will learn today
  - Challenges in code reuse with different data types
  - Introduction to Generics
  - Usage of Wildcards with Generics
  - Functional Interfaces & Lambdas



Motivation for Generics

Why generics and real bugs

- Raw collections accept anything and hide errors until runtime
- Casts clutter code and fail late with ClassCastException.
- Generics add compile time checks and clearer self documenting APIs
- Type parameters eliminate casts and guide usage through signatures

```
List cart = new ArrayList();
cart.add("Apple");
cart.add(199); // accidentally mixing types
for (Object o : cart) {
  String name = (String) o; // fails at runtime
  System.out.println(name.length());
}
```

Activity: spot the risk

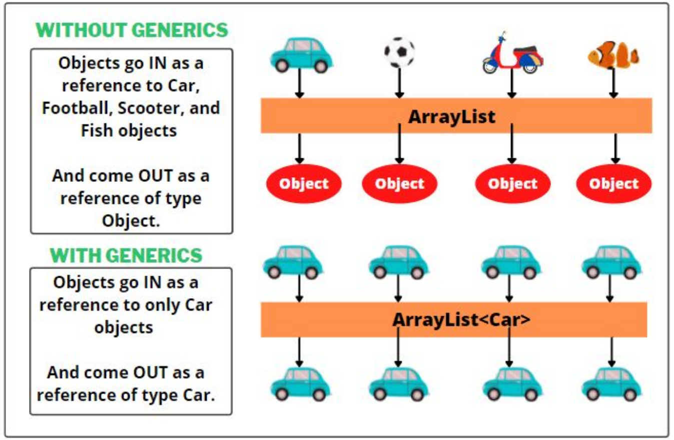
- Goal: design the cart so wrong typed entries cannot be added
- Read both options and pick the safer contract for this context
- State what compile time guarantee your choice provides and why

```
// Option A: parallel lists
List names = new ArrayList();
List prices = new ArrayList();
void add(String n,int p){
  names.add(n);
  prices.add(p);
}
// Option B: wrapper type
class Item {
  String n; int p;
  Item(String n,int p){
    this.n=n;
    this.p=p; }
  }
  List items = new ArrayList();
  void add(Item it) {
    items.add(it);
  }
}
```

Generics: Fundamentals

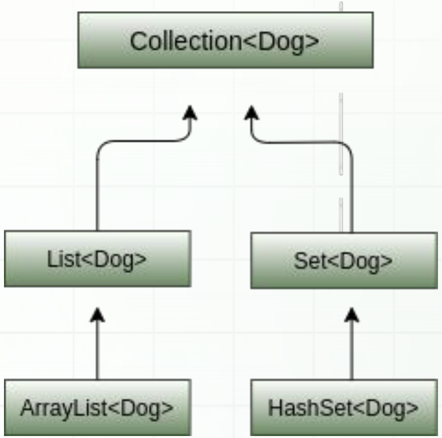
Generic types concept

- A generic type declares a placeholder for a type.
- The compiler enforces one consistent type across params, fields, returns
- Generic methods introduce type parameters scoped to that method
- Treat the signature as a contract stating allowed inputs and outputs



What’s a Generic class?

- A generic class declares a reusable placeholder that becomes concrete once instantiated
- The placeholder becomes a real type at construction time
- Wrong types are rejected by the compiler
- Signatures reveal intended element type to readers and IDE tooling



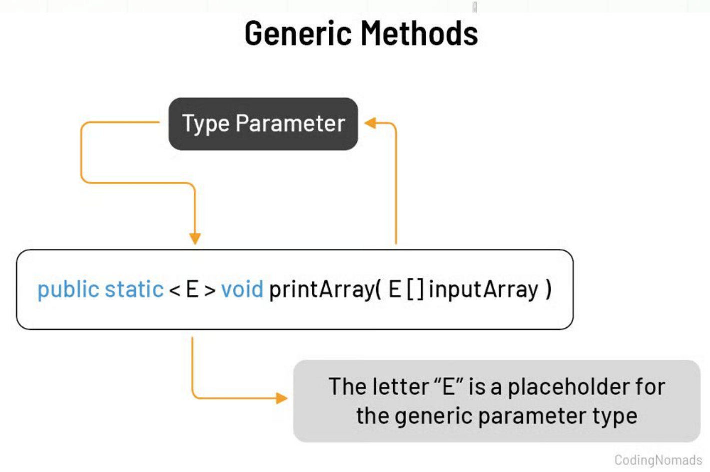
Generic class in practice

```java
class Box<T>{
  private T value;
  public Box(T v){ this.value = v; }
  public T get(){ return value; }
  public void set(T v){ this.value = v; }
}
Box<Double> b = new Box<>(10.5);
Double name = b.get();
```

- Type placeholder T flows through fields and method signatures
- Usage shows compile time rejection of mismatched assignments
- No casts are needed when retrieving typed values
- The pattern scales across many domain types without duplication

Generics in method

- Methods can declare their own type parameter T
- Call sites often infer type arguments automatically
- Generic methods reduce duplication across similar operations
- Bounds arrive later to constrain valid type arguments



Generic method in practice

```
public static <T> T first(List<T> list){
  return list.get(0);
}
List<String> names = List.of("Asha","Ben");
String n = first(names);
```

- T is introduced on the method rather than a class, using <T>
- The call infers T as String from the argument type
- Works for any List without per type overloading
- Such practice encourages small utility methods that stay strongly typed

Activity: build a typed container

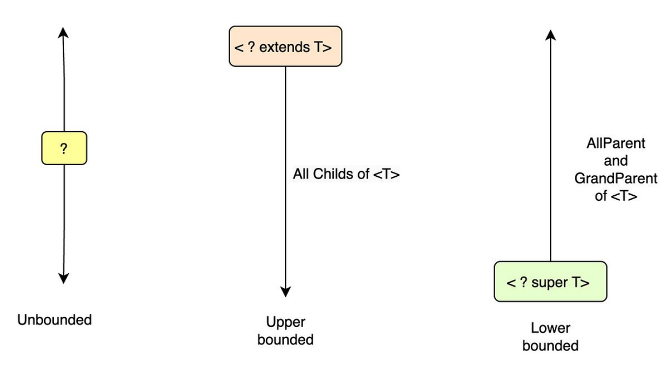
- Goal: define a minimal container that enforces one element type.
- Implement add and first so the API stays type safe
- No casts, no instanceof checks, no raw types

```
class Cart<T>{
  private List<T> items = new ArrayList<>();
  void add(T t){ /* TODO */ }
  T first(){ /* TODO */ return null; }
}
```

Bounds and wildcards

Why bounds on type parameters

- Unbounded generics accept any T, which can permit invalid uses
- Some APIs require a numeric family; others require an ordering
- Bounds express must be a Number or must be Comparable at compile time
- This pushes misuse to compile errors instead of late failures
- We will use upper bounds to accept subtypes when we only read values
- We will use lower bounds to generalize destinations when we write values



Upper bounds with extends

```
static double sum(List<? extends Number> xs) {
  double s = 0.0;
  for(Number n : xs) s += n.doubleValue();
  return s;
}
```

- Number is the common base class for Integer, Double, BigDecimal
- One signature accepts any List of a Number subtype without overloading
- Safe to iterate and read as Number; do not add elements to xs
- Use an upper bound when your method only reads from the list

Lower bounds with super

```
static void addInts(List<? super Integer> dst) {
  dst.add(1);
  dst.add(2);
}
```

- Accepts any List where X is Integer, Number, or Object
- Safe to write Integer values; reading from dst yields Object
- Use a lower bound when your method produces values into a sink
- Prevents accidental insertion of non Integer values

Wildcards in APIs

```
static void printAll(List<?> xs){
  for(Object x : xs) {
    System.out.println(x);
  }
}
```

- Unbounded wildcard allows any element type for reading
- You can read as Object but cannot add elements
- Use bounded wildcards when constraints are needed
- Prefer wildcard when no new type variable is required

Activity: choose the right bound

- For each task, write the parameter type using extends or super
- Explain briefly why your choice is safe

```
// Task A: sum all numbers in a list
static double sum(/* choose bound */ list){ /* body omitted */ }
// Task B: append a few integers into a list
static void fillWithInts(/* choose bound */ dst){ /* body omitted */ }
```

Functional interfaces and lambdas

Functional interfaces concept

- A functional interface has exactly one abstract method to implement
- It is the target type for a lambda at assignment or call sites
- Useful shapes for this lecture: transform price, format name, test condition
- We will start with tiny domain interfaces, then use Comparator at a basic level

Lambda syntax concept

```
class Item {
  String name;
  int price;
  public Item(String n,int p){
    name=n; price=p;
  }
}

@FunctionalInterface interface PriceRule{
  int apply(int price);
}
// Anonymous class vs lambda
PriceRule tenOff1 = new PriceRule(){
  public int apply(int p) { return p - 10; }
};
PriceRule tenOff2 = p -> p - 10;
```

- Both versions implement the same single abstract method apply
- Lambda removes boilerplate while preserving behavior and type
- Target type comes from the variable declared as PriceRule
- Keep bodies short and focused on the small task

Custom functional interface with a lambda

```
public class Item{
  String name;
  int price;
  public Item(String n,int p){
    name=n; price=p;
  }
}

@FunctionalInterface interface ItemTest{
  boolean ok(Item it);
}
int max = 200; // effectively final
ItemTest cheap = it -> it.price <= max;
System.out.println(cheap.ok(new Item("Tea",150)));
```

- One abstract method interface makes the lambda target explicit
- Captured local variables must be effectively final in Java
- Fits real checks like price caps using simple loops later
- No Streams or new APIs required to use the test

Activity: write your first lambda

- Replace the anonymous class with an equivalent lambda using Task
- Rewrite job as a lambda with identical behavior

```
interface Task { void run(); }
Task job = new Task(){
  public void run(){
    System.out.println("Order placed");
  }
};
job.run();
```

Comparator with a lambda

```
Comparator<String> byLen = (a, b) -> Integer.compare(a.length(), b.length());
List<String> names = new ArrayList<>(List.of("Mira","Ben","Asha"));
names.sort(byLen);
System.out.println(names);
```

- Comparator is a single method functional interface for ordering
- The lambda encodes the comparison rule clearly
- Use list.sort with a comparator for simple ordering needs
- Stay at basic ordering and avoid complex rules here

Method references overview

```
interface Printer { void print(String s); }
Printer p = System.out::println;
p.print("Hi");
Comparator<String> byLen = Comparator.comparingInt(String::length);
List<String> names = new ArrayList<>(List.of("Mira","Ben","Asha"));
names.sort(byLen);
```

- Use a method reference when a lambda would just call one method
- Instance and static references reduce noise in trivial cases
- Keep using lambdas when extra logic or data is needed
- Method references integrate with our small interfaces cleanly

That’s for today!
Any questions?