Generics in Java Level 1

Generics

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,neccessary hinds, points to remember,usual mistakes,etc.

start lvl1

Alright Divakar! Let's start **Generics in Java** step by step, **level 1**, like a game. We'll keep it practical, with examples, real-world use cases, hints, points to remember, and common mistakes. 🕹️

Level 1: Basic Generics – Single Type Parameter

1️⃣ Concept

Generics allow classes, interfaces, and methods to operate on **objects of various types** while providing **compile-time type safety**.
Instead of using `Object` and casting, you specify a type parameter `<T>`.

- `T` is a placeholder for the actual type.
- It helps avoid `ClassCastException`.
- Makes code reusable and clean.

2️⃣ Simple Example

```java
// Generic Box class
class Box<T> {
    private T value;  // T is a placeholder type

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        // Box for Integer
        Box<Integer> intBox = new Box<>();
        intBox.set(123);
        System.out.println("Integer Box: " + intBox.get());

        // Box for String
        Box<String> strBox = new Box<>();
```

```
        strBox.set("Hello Generics!");
        System.out.println("String Box: " + strBox.get());
    }
}
```

Explanation

- **`Box<T>`** → T can be replaced with any type.
- **`intBox`** → T becomes **`Integer`**.
- **`strBox`** → T becomes **`String`**.
- No casting needed when retrieving values.

3️⃣ Real-World Use Case

- A shopping cart that can store items of any type:

```
java

Box<Product> cart = new Box<>();
cart.set(new Product("Laptop", 50000));
```

- You can reuse the same class for **`Product`**, **`String`**, **`Integer`**, etc.

4️⃣ Points to Remember

1. Generics work **only with reference types**, not primitives (**`int`**, **`double`** → use **`Integer`**, **`Double`**).
2. Type safety at compile-time prevents runtime errors.
3. Generic classes and methods make code **reusable and readable**.

5️⃣ Common Mistakes

1. Using primitives directly:

```
java

Box<int> box = new Box<>(); // ❌ Error
```

2. Casting defeats the purpose:

```
java

Box box = new Box(); // old-style, not type-safe
Integer value = (Integer) box.get(); // unnecessary cast
```

3. Overusing generic names: use **descriptive names** if possible (**`T`** is okay for single type, **`K`**, **`V`** for maps).

✅ Hints for Level 1

- Think of `<T>` as a "placeholder" for a real type you will specify later.
- Use Generics **whenever you want to create reusable containers or utilities**.

If you want, **next level (Level 2)** will cover **Generic Methods and Multiple Type Parameters**, which is like leveling up in the game. 🎮

Do you want me to proceed to **Level 2**?

i want each level with possible more different examples for more clear understand

Perfect! We'll make each level **super clear** with **multiple examples**, explanations, real-world scenarios, hints, common mistakes—like "leveling up" step by step in a game. Let's **expand Level 1** first with more examples and use cases before moving to Level 2. 🎮

Level 1: Basic Generics – Single Type Parameter (Expanded)

Example 1: Box class (already discussed)

```java

class Box<T> {
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>();
        intBox.set(10);
        System.out.println(intBox.get());

        Box<String> strBox = new Box<>();
        strBox.set("Generics are fun!");
        System.out.println(strBox.get());
    }
}
```

✅ Key points

- T is replaced by the specified type (`Integer`, `String`).
- No casting needed.
- Type safety at compile-time.

Example 2: Generic Pair

```java
class Pair<T> {
    private T first;
    private T second;

    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }
    public T getSecond() { return second; }
}

public class Main {
    public static void main(String[] args) {
        Pair<String> namePair = new Pair<>("Alice", "Bob");
        System.out.println(namePair.getFirst() + " & " + namePair.getSecond());

        Pair<Integer> numberPair = new Pair<>(1, 2);
        System.out.println(numberPair.getFirst() + " & " + numberPair.getSecond());
    }
}
```

💡 **Use case:** Storing pairs of same type values, e.g., coordinate points, name pairs, IDs.

Example 3: Generic Method (Level 1 Intro)

You can make methods generic **without making the class generic**.

```java
class Utils {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        Integer[] intArr = {1, 2, 3};
        String[] strArr = {"Java", "Generics"};

        Utils.printArray(intArr);
        Utils.printArray(strArr);
    }
}
```

💡 **Use case:** Utility methods to work on multiple types (like printing arrays, swapping elements, etc.)

Example 4: Real-world scenario – Shopping Cart

```java
class Cart<T> {
    private T item;
```

```
    public void addItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}

class Product {
    String name;
    double price;

    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    public String toString() {
        return name + " ($" + price + ")";
    }
}

public class Main {
    public static void main(String[] args) {
        Cart<Product> productCart = new Cart<>();
        productCart.addItem(new Product("Laptop", 80000));
        System.out.println(productCart.getItem());

        Cart<String> giftCart = new Cart<>();
        giftCart.addItem("Gift Voucher");
        System.out.println(giftCart.getItem());
    }
}
```

💡 **Use case:** Shopping carts, inventory, reusable storage containers.

Hints & Points to Remember (Level 1)

1. **Generics only work with reference types** → use `Integer` instead of `int`, `Double` instead of `double`.
2. Avoid raw types:

```
java

Box box = new Box(); // ❌ Not recommended
```

3. Generics **provide compile-time safety** → prevents `ClassCastException`.
4. Generic type names convention:
   - `T` → Type
   - `E` → Element (for collections)
   - `K` → Key, `V` → Value (for maps)
5. **Reusable code**: write once, use for multiple types.

Common Mistakes

1. Using primitive types directly:

```java
Box<int> box = new Box<>(); // ❌ Error
```

2. Mixing generic types:

```java
Box<String> box = new Box<>();
box.set(123); // ❌ Compile-time error
```

3. Overcomplicating with too many type parameters at Level 1. Keep it simple first.

If you want, **Level 2** will introduce:

- **Multiple type parameters `<T, U>`**
- **Generic methods in detail**
- **Real-world examples with Maps and utility functions**

Do you want me to **proceed to Level 2 now**? Or should I add **even more Level 1 examples** for deeper clarity?

next

Awesome! 🎮 Let's **Level 2**: We're "leveling up" in Generics.

At Level 2, we'll cover:

- **Multiple Type Parameters `<T, U>`**
- **Generic Methods in detail**
- **Real-world examples**

Level 2: Multiple Type Parameters & Generic Methods

1️⃣ Concept

- Classes or methods can have **more than one generic type parameter**.
- Useful when you want to work with **two or more types in a single class or method**.

Syntax:

```java
class Pair<T, U> {
    private T first;
```

```
    private U second;
}
```

2️⃣ Example 1: Generic Pair with Two Types

```
java

class Pair<T, U> {
    private T first;
    private U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() { return first; }
    public U getSecond() { return second; }

    public void display() {
        System.out.println("First: " + first + ", Second: " + second);
    }
}

public class Main {
    public static void main(String[] args) {
        Pair<String, Integer> nameAge = new Pair<>("Alice", 25);
        nameAge.display();  // First: Alice, Second: 25

        Pair<Integer, Double> idSalary = new Pair<>(101, 50000.50);
        idSalary.display();  // First: 101, Second: 50000.5
    }
}
```

💡 **Use case:** Storing **key-value pairs** like student name + grade, product + price.

3️⃣ Example 2: Generic Method with Multiple Types

```
java

class Utils {
    public static <T, U> void printPair(T first, U second) {
        System.out.println("First: " + first + ", Second: " + second);
    }
}

public class Main {
    public static void main(String[] args) {
        Utils.printPair("Alice", 25);
        Utils.printPair(101, 50000.50);
        Utils.printPair("Product", 499.99);
    }
}
```

💡 **Use case:** Generic utility functions like **swap values**, **print any combination of types**, or **compare two objects**.

4️⃣ Example 3: Real-world scenario – Product and Quantity

```java
class Item<T, U> {
    private T product;
    private U quantity;

    public Item(T product, U quantity) {
        this.product = product;
        this.quantity = quantity;
    }

    public void showItem() {
        System.out.println("Product: " + product + ", Quantity: " + quantity);
    }
}

public class Main {
    public static void main(String[] args) {
        Item<String, Integer> grocery = new Item<>("Apple", 10);
        grocery.showItem();

        Item<String, Double> electronics = new Item<>("Laptop", 2.5); // weight
        electronics.showItem();
    }
}
```

💡 **Use case:** Inventory system: store **product** with **quantity**, **weight**, or **price**.

5️⃣ Hints & Points to Remember

1. You can have **any number of type parameters**: `<T, U, V>` etc.
2. **Generic methods** don't need a generic class to exist.
3. **Multiple types** are great for **utility methods, key-value storage, and heterogeneous containers**.
4. Always specify **types at object creation**; otherwise raw types remove safety.

6️⃣ Common Mistakes

1. Forgetting to declare generic types for methods:

```java
public static void print(T first, U second) {} // ❌ Error
```

✅ Correct:

```java
public static <T, U> void print(T first, U second) {}
```

2. Mixing type parameters incorrectly:

```java
Pair<String, Integer> pair = new Pair<>(25, "Alice"); // ❌ Type mismatch
```

3. Overcomplicating with too many type parameters unnecessarily. Keep it readable.

🎯 **Level 2 Complete**:
You now know:

- Multiple type parameters in classes and methods
- How to use them for key-value pairs and utilities
- Real-world examples in shopping carts, inventory, and utility functions

---

If you want, **Level 3** will cover:

- **Bounded Generics** (`<T extends SomeClass>`)
- **Wildcards** (`?`, `? extends`, `? super`)
- Advanced use cases with **collections**

Do you want me to proceed to **Level 3** next?

next

Perfect! 🎮 Let's **Level 3: Bounded Generics & Wildcards** — this is where Generics get powerful in **real-world scenarios**, especially with **collections and constraints**.

Level 3: Bounded Generics & Wildcards

1️⃣ Concept: Bounded Generics

Sometimes, you **want to restrict the types** a generic can accept.

- **Upper Bound:** `<T extends ClassName>` → T can be ClassName or its **subclasses**
- **Lower Bound:** `<T super ClassName>` → T can be ClassName or **superclasses** (used mostly in wildcards)

✅ Useful when you want **type safety with specific hierarchy**.

2️⃣ Example 1: Upper Bounded Generic

```java
// Base class
class Animal {
    void sound() { System.out.println("Some animal sound"); }
}

// Subclasses
```

```java
class Dog extends Animal {
    void sound() { System.out.println("Woof!"); }
}

class Cat extends Animal {
    void sound() { System.out.println("Meow!"); }
}

// Generic class restricted to Animal or its subclasses
class Cage<T extends Animal> {
    private T animal;

    public Cage(T animal) { this.animal = animal; }

    public void makeSound() {
        animal.sound();
    }
}

public class Main {
    public static void main(String[] args) {
        Cage<Dog> dogCage = new Cage<>(new Dog());
        dogCage.makeSound(); // Woof!

        Cage<Cat> catCage = new Cage<>(new Cat());
        catCage.makeSound(); // Meow!

        // Cage<String> strCage = new Cage<>("Hello"); // ❌ Error, String not an Animal
    }
}
```

💡 **Use case:** Zoos, pet shops, or any container restricted to certain types.

3️⃣ Example 2: Bounded Generic Method

```java
java


class Utils {
    // Accepts numbers only (Integer, Double, etc.)
    public static <T extends Number> double sum(T a, T b) {
        return a.doubleValue() + b.doubleValue();
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(Utils.sum(10, 20));        // 30.0
        System.out.println(Utils.sum(15.5, 20.3));    // 35.8
        // Utils.sum("Hello", "World"); // ❌ Error
    }
}
```

💡 **Use case:** Generic math functions, numeric calculations, or type-safe computations.

4️⃣ Example 3: Wildcards in Generics

- `?` → unknown type
- `? extends Class` → upper-bounded wildcard (read-only)
- `? super Class` → lower-bounded wildcard (write-safe)

```java
import java.util.*;

public class Main {
    public static void printAnimals(List<? extends Animal> animals) {
        for (Animal a : animals) {
            a.sound();
        }
    }

    public static void main(String[] args) {
        List<Dog> dogs = Arrays.asList(new Dog(), new Dog());
        List<Cat> cats = Arrays.asList(new Cat(), new Cat());

        printAnimals(dogs); // Woof! Woof!
        printAnimals(cats); // Meow! Meow!
    }
}
```

💡 **Use case:** Collections with **mixed subclasses**.

- You can **read** safely from `? extends` lists.
- You can **write** safely to `? super` lists.

5️⃣ Hints & Points to Remember

1. Use **bounded generics** to **limit allowed types**.
2. `extends` → read-only access; `super` → write-safe access.
3. Wildcards are **mainly useful with collections**.
4. Avoid raw types — they **remove type safety**.

6️⃣ Common Mistakes

1. Forgetting upper bounds:

```java
Cage<String> cage = new Cage<>("Hello"); // ❌ Error
```

2. Trying to add elements to `List<? extends Animal>`:

```java
List<? extends Animal> list = new ArrayList<Dog>();
list.add(new Dog()); // ❌ Error
```

3. Confusing `extends` vs `super`:
   - `extends` → read from list safely
   - `super` → write to list safely

🎯 **Level 3 Complete**:
You now know:

- Bounded Generics (`extends`)
- Generic methods with constraints
- Wildcards (`?`, `? extends`, `? super`)
- Real-world examples in animals, products, and collections

If you want, **Level 4** can cover:

- **Generics with Collections in depth** (List, Map, Set)
- **Advanced use cases like sorting, filtering, and reusable utilities**

Do you want to proceed to **Level 4**?