Raw Types vs. Wildcards

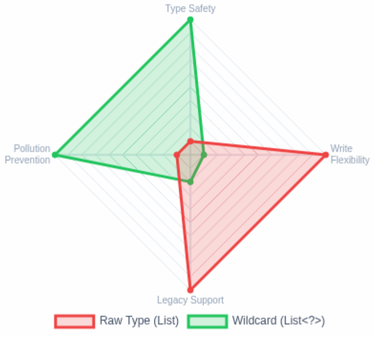
A showdown between the legacy "Escape Hatch" and the modern "Type-Safe Abstraction". Understand why `List` invites disaster while `List<?>` enforces discipline.

? THE UNBOUNDED WILDCARD

1. Two Philosophies: Unsafe vs. Safe

Java offers two ways to handle "unknown" types, but they behave radically differently. The **Raw Type** (`List`) disables generic checks for backward compatibility, often creating hidden runtime bombs. The **Unbounded Wildcard** (`List<?>`) acknowledges the unknown but strictly prohibits unsafe operations.

Safety Profile Comparison



Visualizing the trade-off: Raw types offer high write flexibility at the cost of zero safety.

LEGACY

The Raw Type (List)

"I'm pretending Generics don't exist."
Used primarily for pre-Java 5 compatibility. It turns off type checking entirely, allowing *Heap Pollution*.

- ✕ Unsafe: No type checks.
- ✓ Allows adding ANY Object.
- Risk: ClassCastException at runtime.

MODERN

The Wildcard (List<?>)

"I don't know the type, so I won't touch it."
The type-safe abstraction. It is universal but acts as a read-only view to prevent corruption.

- ✓ Safe: Strict rule enforcement.
- Adding Forbidden (except null).
- Universal: Points to any List<T>.

2. The Danger: Heap Pollution

Raw types allow you to insert incompatible objects into a list. This corruption is silent at the moment of insertion (the "Write") but causes a crash later during retrieval (the "Read").

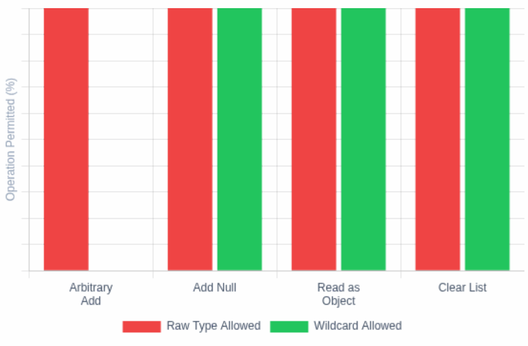
Main.java — Unsafe Mode

```
List<String> strings = new ArrayList<>();
List raw = strings; // Raw Type Alias

// 1. The Silent Corruption
raw.add(10); // Allowed! No Error.
raw.add(true); // Allowed! No Error.

// 2. The Delayed Crash
String s = strings.get(1); // BOOM! ClassCastException
```

Operations: Raw vs. Wildcard



Comparison of permitted actions. Note how Wildcards block 'Add' to ensure safety.

3. When to use List<?>

Use Unbounded Wildcards when your method logic relies only on `Object` methods or the internal structure of the container, effectively treating the data as "opaque".

A. Generic Utilities

Read-only processing where `Object.toString()` is sufficient.

```
public void printList(List<?> list) { ... }
```

B. Container Ops

Structural changes like `clear()`, `size()`, or `shuffle()`.

```
public void shuffle(List<?> list) { ... }
```

C. Class Literals

Fetching class metadata where the specific type T is irrelevant.

```
Class<?> c = obj.getClass();
```

Summary Breakdown

Feature	Raw Type (List)	Wildcard (List<?>)
Safety Level	⚠ Unsafe (Disabled)	Safe (Strict)
Add Operation	Allowed (Pollution Risk)	Forbidden (Except null)
Read Return Type	Object	Object
Primary Use Case	Legacy Compatibility	Generic Utilities