Java Generics

Syntax, Intuition & Architecture

Generics move the risk from Runtime crashes to Compile-time safety. Master the "Alphabet Soup" of syntax, understand the "Search-and-Replace" mechanics of Erasure, and leverage the PECS rule for flexible API design.

<T>

The Architect's Blueprint

1. The "Why": The Safety Shift

Before Java 5, collections were "Boxes of Mystery" holding Objects. This relied on human memory, often leading to **Runtime ClassCastExceptions**. Generics enforce strict type labels, shifting errors to the compiler where they are cheap to fix.

Pre-Generics (Legacy)

Relies on manual casting. Risk of "Heap Pollution" is high because `add(Object)` accepts anything.

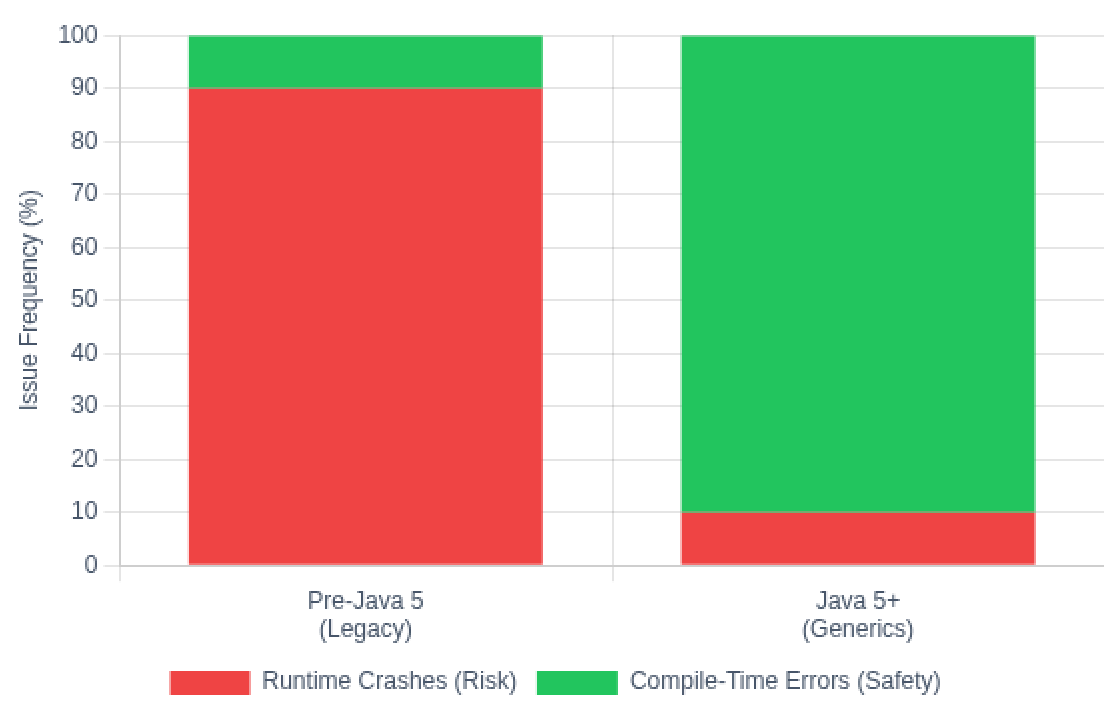
```
List raw = new ArrayList();
raw.add("Hello");
// CRASH at Runtime!
Integer i = (Integer) raw.get(0);
```

With Generics (Modern)

Compiler acts as a security guard. Incompatible types are rejected instantly.

```
List<String> safe = new ArrayList<>();
safe.add("Hello");
// Compiler ERROR! Safe.
// Integer i = safe.get(0);
```

Risk Profile: Runtime vs. Compile-Time



Comparison of error discovery timing.

2. Syntax & Intuition: Where does <T> go?

The syntax changes based on whether you are **defining** a blueprint (Class), **granting permission** (Method), or **using** a reference (Variable).

The Class Label

DEFINITION PHASE

When defining a class, the <T> is the **Identity Label**. It tells the compiler "This box deals with type T".

```
public class Box<T>
```

- ✓ Placed immediately after Class Name.
- ✓ Acts as a placeholder for fields.

The Permission Slip

METHOD PHASE

Static methods don't know the class's T. You must grant a **Permission Slip** before the return type.

```
public <T> void print(T item)
```

- ✓ Placed **before** Return Type.
- ✓ Required for type inference to work.

The Flexible Ref

VARIABLE PHASE

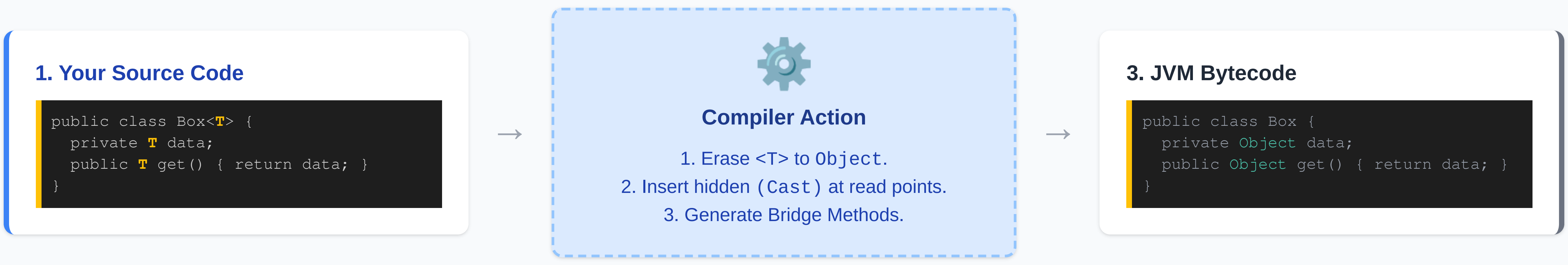
When using a variable, you don't define T, you **reference** it. Use Wildcards for flexibility.

```
List<? extends Number> list
```

- ✓ Used for arguments or fields.
- ✓ Defines read/write capabilities.

3. Under the Hood: Type Erasure

Generics are a "Compile-time Trick". The JVM is blind to them. The compiler performs a massive **Search-and-Replace** operation to maintain backward compatibility.



1. Your Source Code

```
public class Box<T> {
  private T data;
  public T get() { return data; }
}
```

→

Compiler Action

1. Erase <T> to `Object`.
2. Insert hidden `(Cast)` at read points.
3. Generate Bridge Methods.

→

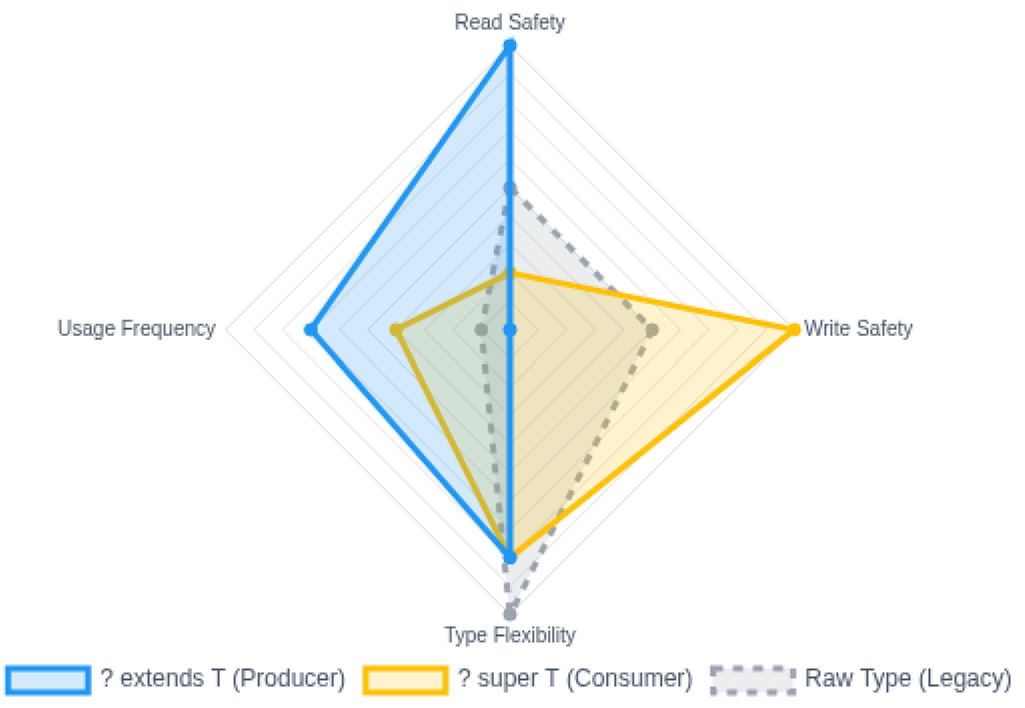
3. JVM Bytecode

```
public class Box {
  private Object data;
  public Object get() { return data; }
}
```

4. The PECS Rule: Handling Invariance

Generics are **Invariant** (`List<Integer>` is NOT `List<Number>`). To handle hierarchy, we use Wildcards governed by **P**roducer **E**xtends, **C**onsumer **S**uper.

Wildcard Capabilities



Shows what is legally allowed for each wildcard type.

PRODUCER `<? extends T>`

Goal: Reading Data.
Analogy: "I need a list that provides Numbers."
🚫 CANNOT ADD (except null)

CONSUMER `<? super T>`

Goal: Writing Data.
Analogy: "I need a list that can hold Integers."
✓ CAN ADD T

The Forbidden Zones (Due to Erasure)

No new T()
JVM doesn't know constructor

No instanceof T
T is Object at runtime

No catch(T)
Cannot verify exception type

No Generic Arrays
Arrays need Reified types

Java Generics Visualization

Based on Type Erasure, Invariance, and PECS Principles