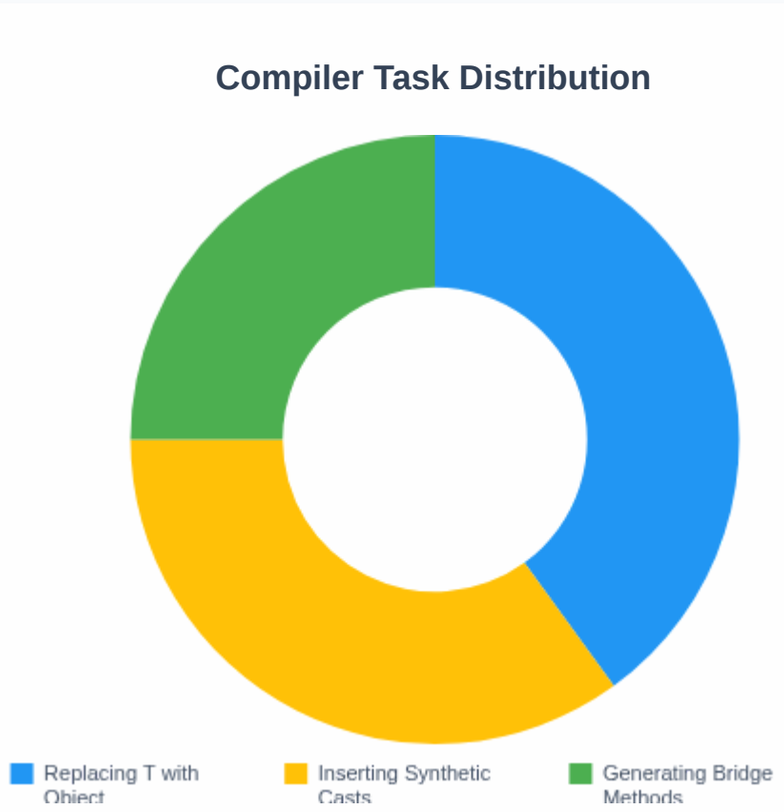
Type Erasure Deep Dive

The JVM's "Search-and-Replace" trick that powers Java Generics. Understanding how the compiler sacrifices runtime type information to maintain backward compatibility.

<T>
BECOMES
Object

1. The Erasure Mechanism

Type Erasure isn't magic; it's a systematic three-step process performed by the compiler. It transforms your generic code into standard bytecode that any JVM (even pre-Java 5) can understand.



Breakdown of compiler actions during the erasure process.

1. Replace Type Parameters

Every <T> is deleted. Unbounded types become `Object`. Bounded types (`<T extends Number>`) become their bound (`Number`).

2. Insert Synthetic Casts

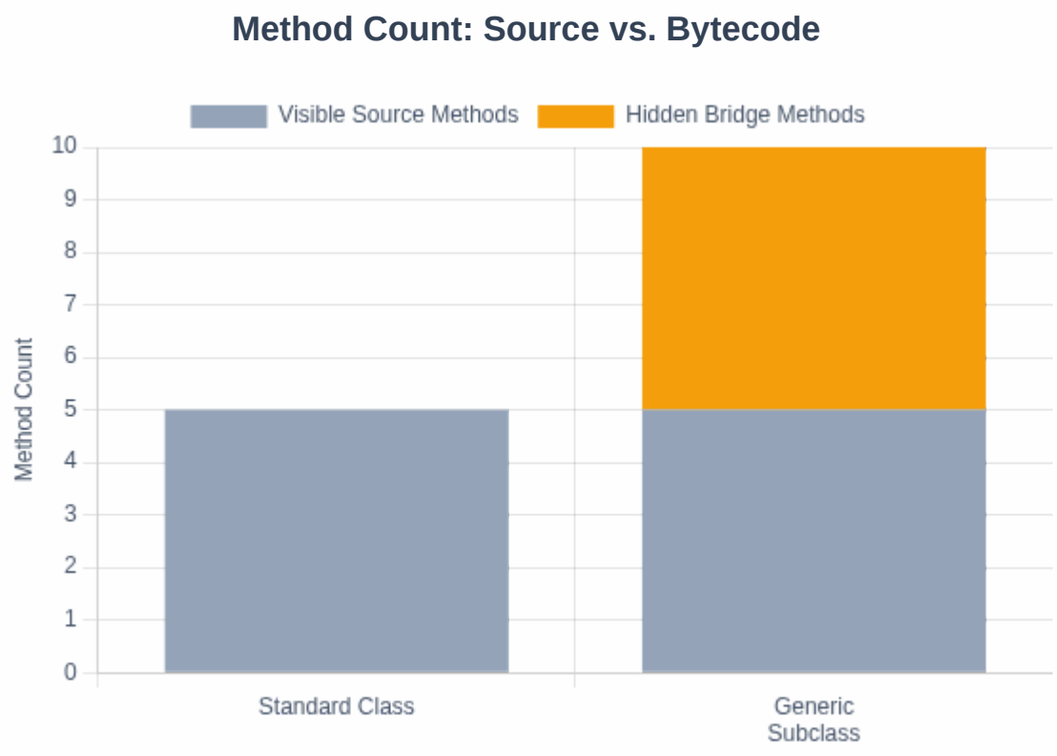
Since the data is stored as `Object`, the compiler inserts a hidden `checkcast` instruction whenever you read data back into a typed variable.

3. Generate Bridge Methods

To fix polymorphism breaks caused by signature mismatches, the compiler generates synthetic "Bridge Methods" that delegate to your actual logic.

2. The Hidden Costs: Bridges & Casts

Erasure isn't free. It adds invisible instructions to your bytecode. Implementing a generic interface often results in double the number of methods in the compiled class file to maintain polymorphism.



Extending a generic class creates invisible "Bridge Methods" in the bytecode.

THE "HIDDEN" CAST LOGIC

```
// Source Code
List<String> list = new ArrayList<>();
String s = list.get(0);

// What JVM Actually Executes
List list = new ArrayList();
// Compiler inserts this checkcast:
String s = (String) list.get(0);
```

Why this matters?

This hidden cast is why "Heap Pollution" is dangerous. If you sneak an Integer into a raw List, the crash happens at the **read** line (`get()`), not the **write** line.

3. Signature & Interface Collisions

Because generic types vanish, distinct source code signatures can collapse into identical bytecode signatures, causing compile-time errors.

Overloading Failure

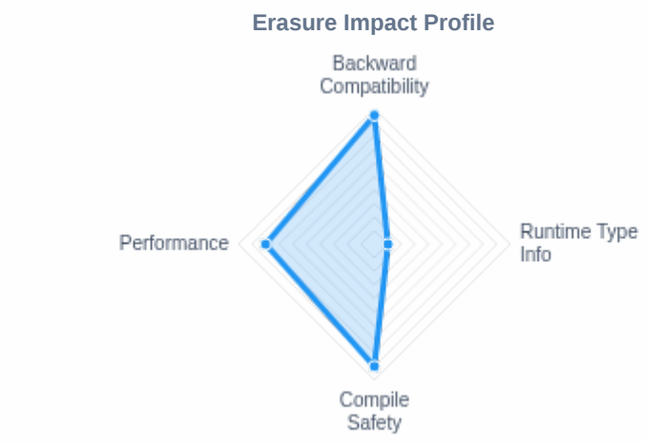
```
void print(List<String> l)
void print(List<Integer> l)
```

Result: Both become `print(List l)`.
The JVM forbids two methods with the exact same signature.

Interface Collision

```
implements Comparable<String>,
Comparable<Integer>
```

Result: Both erase to `Comparable`.
Bridge methods would collide trying to bridge `Object` to both `String` and `Integer`.



4. The Forbidden Zones

Operations that rely on runtime type information are impossible because that information is deleted.

No new T()

JVM sees `new Object()`. It doesn't know which constructor to call.

No instanceof T

`T` is erased to `Object`. The check would be uselessly broad.

No catch(T)

Exception handling is a runtime feature. The JVM can't distinguish erased types.

No Generic Arrays

Arrays are Reified (strict). Generics are Erased. Mixing them breaks memory safety.

Java Internals: Type Erasure Infographic

Visualizing the "Search-and-Replace" Architecture of the Java Compiler