

Advanced Generics



Jesper de Jong

Software Architect

@jesperdj www.jesperdj.com



This Module

Generic **types**

Type parameters
Parameterized types
Type arguments

Generic **methods**

Raw types

Bounded type parameters

Generics and **inheritance**

Generics and **arrays**

Wildcards

Unbounded
Upper bounded
Lower bounded

Type **erasure**



Explanation: Defining Generic Types



Generic Types

```
public class LeafNode<T> implements TreeNode<T> { ... }
```



Type parameter



Generic Types

```
public class LeafNode<T> implements TreeNode<T> { ... }
```

```
var three = new LeafNode<Integer>(3);
```



Type argument



Generic Types

```
public class LeafNode<T> implements TreeNode<T> { ... }
```

```
var three = new LeafNode<Integer>(3);
```

Instantiating a generic type into a **parameterized type**



Generic Types

```
public class LeafNode<T> implements TreeNode<T> { ... }
```

```
var three = new LeafNode<Integer>(3);
```

Creating an **instance** (object) of a **parameterized type**



Value Level and Type Level

Define methods and constructors
with **value parameters**
(int a, String b)



Define generic types
with **type parameters**
<K, V>

Call methods and constructors
with **arguments** = actual values
(23, "Hello")



Supply **type arguments** = actual
types for the type parameters
<Integer, String>

Instantiate a type into
an **object**



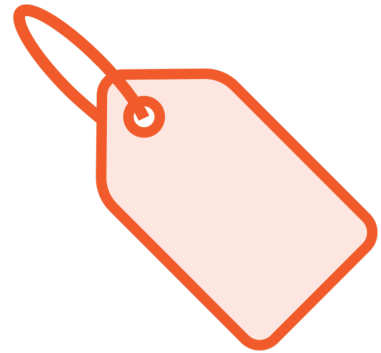
Instantiate a generic type into
a **parameterized type**

Terminology

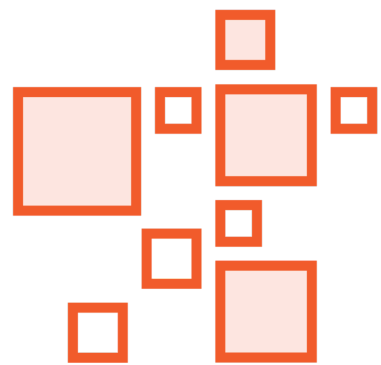
A **generic type** is a type with **type parameters**, which is instantiated into a **parameterized type** by supplying **type arguments** to fill in its type parameters.



More About Generic Types



Single capital letter names for type parameters are a widely accepted **convention**



You can not use **primitive types** as type arguments



Anonymous inner classes, enums and exception classes cannot have type parameters



Use Cases for Generic Types

Generic data structures

Code reuse

Generic abstract superclass

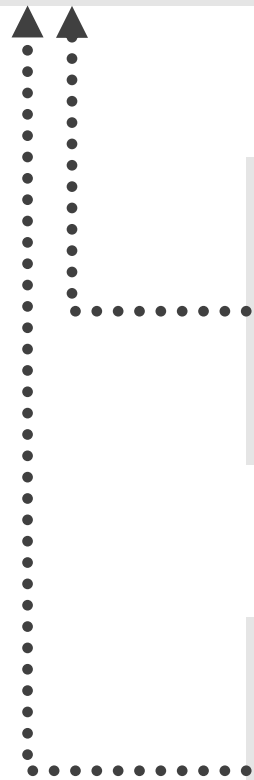


Use Cases for Generic Types

```
abstract class Screen<T>
```

```
class BusinessContractScreen  
    extends Screen<BusinessContractOptions>
```

```
class PersonalContractScreen  
    extends Screen<PersonalContractOptions>
```



Raw Types



Raw Types

```
List objects = new ArrayList(); // No type arguments <...>
```



Raw Types

```
List objects = new ArrayList(); // No type arguments <...>
```

```
objects.add("Hello");
```

```
objects.add(123);
```



Raw Types

```
List objects = new ArrayList(); // No type arguments <...>

objects.add("Hello");
objects.add(123);

String text = (String) objects.get(1); // ClassCastException
```



Avoid using raw types.
They only exist for
backward compatibility.



Wildcards



What is a Wildcard?

A **wildcard** is a way to refer to a **family of types**

?

Unbounded wildcard

? extends SomeType

Upper bounded wildcard

? super SomeType

Lower bounded wildcard

Used to declare **wildcard parameterized types**



Wildcard Parameterized Types

Generic type



Parameterized type

`List<T>`

Generic type



`List<String>`

Concrete parameterized type

`List<T>`

Generic type



`List<?>`

Wildcard parameterized type



Wildcard Parameterized Types

```
List<String>
```

Concrete parameterized type

A single, fully defined type

```
List<?>
```

Wildcard parameterized type

Not a single type

Matches a family of types



Matching Wildcard Parameterized Types

List<?>

... List<String>

..... List<Integer>

..... List<Map<?, ?>>

List<? extends Animal>

... List<Dog>

..... List<Cat>

..... List<Animal>



Using Wildcards in Practice



Using Wildcards in Practice

Defining methods that take **parameters of **parameterized types****

Avoid unnecessary restrictions



Example: Collections.copy()

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```



Example: Collections.copy()

```
public static <T> void copy(List<T> dest, List<T> src)
```

Requires dest and src to have the exact same element type

```
List<Dog> dogs = List.of(new Dog("Daisy"), new Dog("Lucky"));  
List<Animal> animals = new ArrayList<>();
```



Example: Collections.copy()

```
public static <T> void copy(List<T> dest, List<T> src)
```

Requires dest and src to have the exact same element type

```
List<Dog> dogs = List.of(new Dog("Daisy"), new Dog("Lucky"));  
List<Animal> animals = new ArrayList<>();  
copy(animals, dogs);
```



Example: Collections.copy()

```
public static <T> void copy(List<T> dest, List<T> src)
```

Requires dest and src to have the exact same element type

```
List<Dog> dogs = List.of(new Dog("Daisy"), new Dog("Lucky"));  
List<Animal> animals = new ArrayList<>();  
copy(animals, dogs); // Error: Type mismatch
```



Example: Collections.copy()

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```



```
List<Dog> dogs = List.of(new Dog("Daisy"), new Dog("Lucky"));  
List<Animal> animals = new ArrayList<>();  
copy(animals, dogs); // OK
```

Example: Collections.copy()

```
public static <T> void copy(List<? super T> dest, List<? extends T> src)
```



Upper bounded wildcards for **in** parameters

Lower bounded wildcards for **out** parameters

No wildcard for a parameter that is both **in** and **out**



Unbounded Wildcard Parameters

```
public static int size(List<?> list)
```



Unbounded wildcard if the method
does not need to know what the wildcard stands for



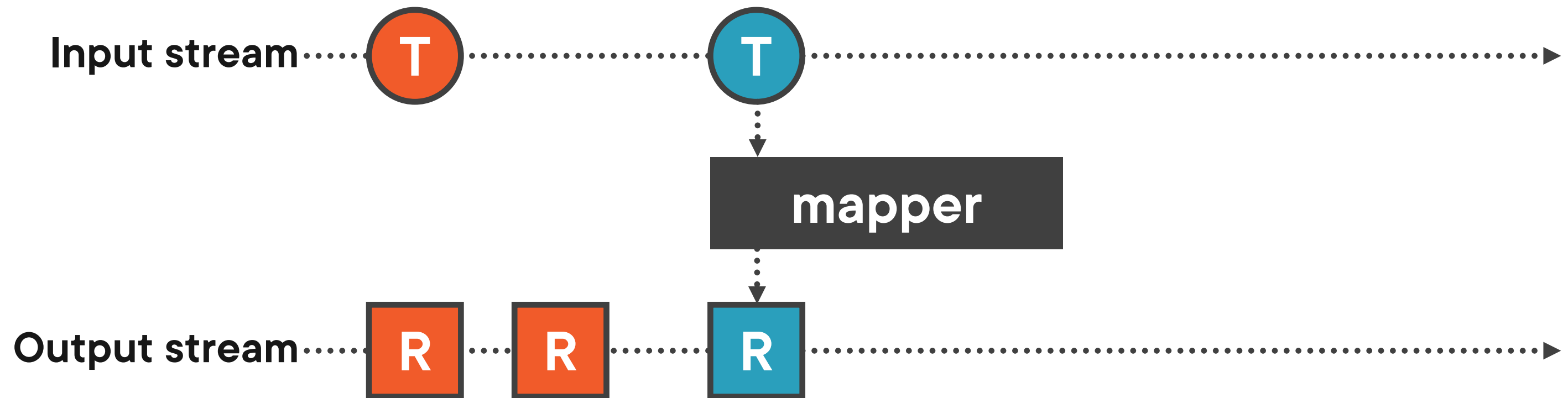
Example: Stream.flatMap()

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```



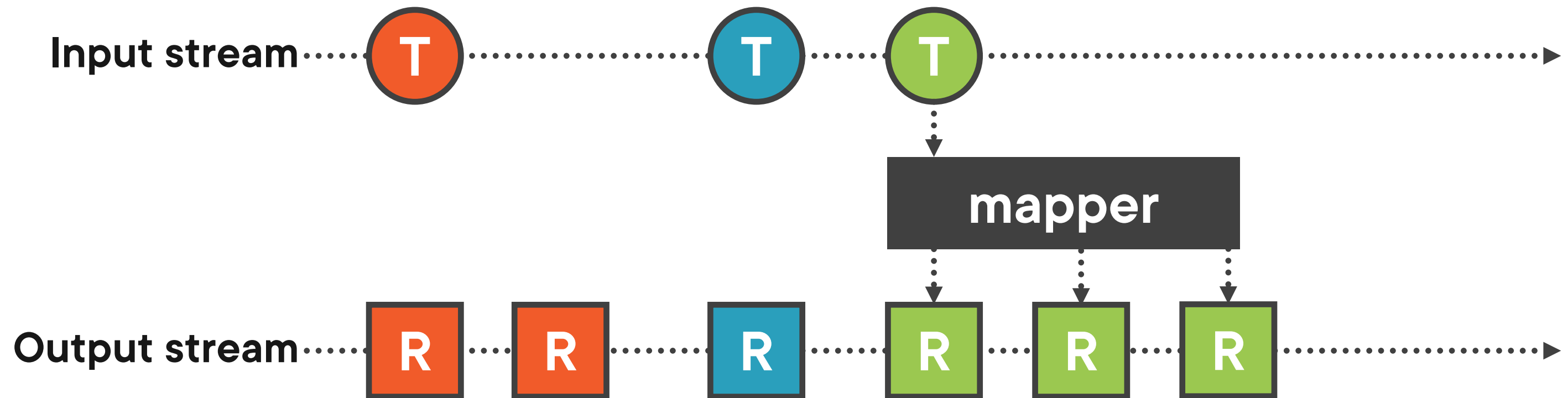
Example: Stream.flatMap()

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```



Example: Stream.flatMap()

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```



One-to-many transformation of a stream of T into a stream of R



Example: Stream.flatMap()

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```



A function that takes an **? super T**
and returns an **? extends Stream<? extends R>**

T - Type of the elements of the Stream that flatMap is called on

R - Type of the elements of the Stream that flatMap returns



Example: Stream.flatMap()

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```



A function that takes an **? super T**
and returns an **? extends Stream<? extends R>**

Upper bounded wildcards for **in** parameters

Lower bounded wildcards for **out** parameters



Example: Stream.flatMap()

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
```



Avoid using wildcards in the **return type** of a method



Practice Reading Wildcard Method Signatures

```
class java.util.Collections
```

```
interface java.util.Comparator
```

```
interface java.util.stream.Stream
```

```
class java.util.stream.Collectors
```



Type Erasure



Type Erasure

Generics are a **compile-time only feature**

**Generic and parameterized types,
type parameters and type arguments do not exist at runtime**

**Type parameters are replaced by
Object or the leftmost bound**

Type arguments are discarded

**Parameterized types
are replaced by raw types**

**Type casts are added
where necessary**



Limitations Caused by Type Erasure



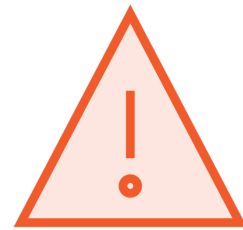
Limitations Caused by Type Erasure

You cannot use `primitive types` as type arguments
Because primitive types are not objects



Limitations Caused by Type Erasure

It's not possible to create
a **new instance** of a **type parameter**



```
new T(); // Error
```

```
Class<T> cls = ...;  
var obj = cls.getDeclaredConstructor().newInstance();
```



Limitations Caused by Type Erasure

instanceof does not work with **non-reifiable types**

Non-reifiable type **A type for which type information is lost during type erasure**

Parameterized types with at least one concrete or bounded wildcard type argument



```
obj instanceof List<String> // Error
```



Limitations Caused by Type Erasure

instanceof does not work with **non-reifiable types**

Non-reifiable type **A type for which type information is lost during type erasure**

Parameterized types with at least one concrete or bounded wildcard type argument

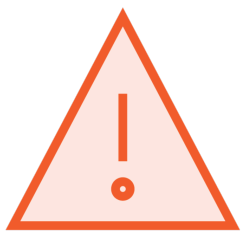


```
obj instanceof List<?> // OK
```



Limitations Caused by Type Erasure

No **class literals** for parameterized types



```
Class<?> cls = List<String>.class; // Error
```



Limitations Caused by Type Erasure

Operations where **type safety** cannot
be guaranteed cause **unchecked** warnings



Limitations Caused by Type Erasure

You cannot **overload methods**
with the same **method signature** after type erasure



```
void print(List<String> strings)  
void print(List<Integer> integers)
```



Limitations Caused by Type Erasure

You cannot **overload methods**
with the same **method signature** after type erasure



```
void print(List strings)  
void print(List integers)
```

After type erasure



Heap Pollution

Separately compiled source files

PetStore.java

```
public List<Dog> getPets() {  
    return List.of(  
        new Dog("Daisy"),  
        new Dog("Lucky"));  
}
```

Application.java

```
List<Dog> pets = petStore.getPets();
```

Heap Pollution

Separately compiled source files

PetStore.java

```
public List<Dog> getPets() {  
    return List.of(  
        new Dog("Daisy"),  
        new Dog("Lucky"));  
}
```

Application.java

```
List<Cat> pets = petStore.getPets();
```



Summary



Defining generic types and methods

- Generic types and type parameters
- Parameterized types and type arguments

Bounded type parameters

Raw types

Generics and inheritance

- Generic types are invariant



Summary



Wildcards

- Wildcard parameterized types
- Unbounded: ?
- Upper bounded: ? extends Type
- Lower bounded: ? super Type

Type erasure

Heap pollution

Generics and arrays

Generics and varargs



Wildcards

A wildcard stands for a **specific** but **unknown** type

```
List<?>
```

A list of objects of a specific unknown type

**You cannot add elements to this list
because the type is unknown**



Up Next: Lambda Expressions
and Method References

