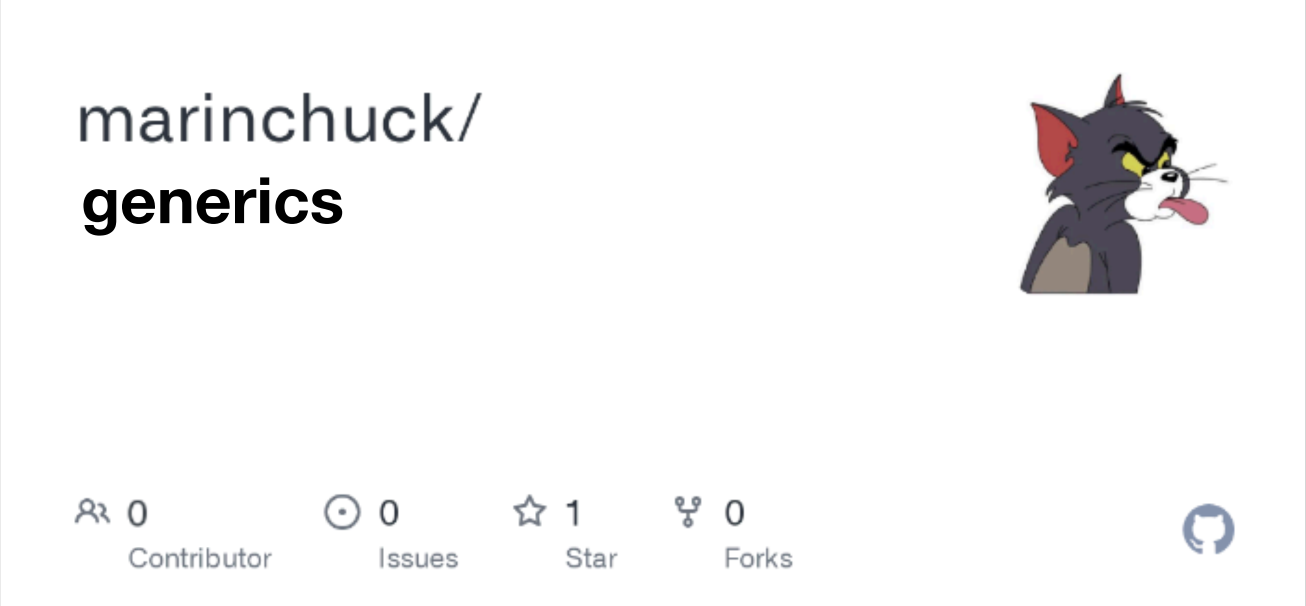
Generics

by: Tamas Marincsak

Table of Contents



marinchuck/
generics

0 Contributor

0 Issues

1 Star

0 Forks

What do we win with generics?

- no need for ~~explicit casting~~
- type checking in **compile time** ~~(ClassCastException)~~
- (enabling programmers to implement **generic algorithms**)

Introduction

- introduced by *JDK 5*
- new syntactic elements (`?`, `<>`)
- rewrite core Java classes, methods, etc.
- can be ***class***, ***interface***, ***method***, ***constructor***

1. Class

A class is generic if it declares one or more type variables. These type variables are known as the type parameters.

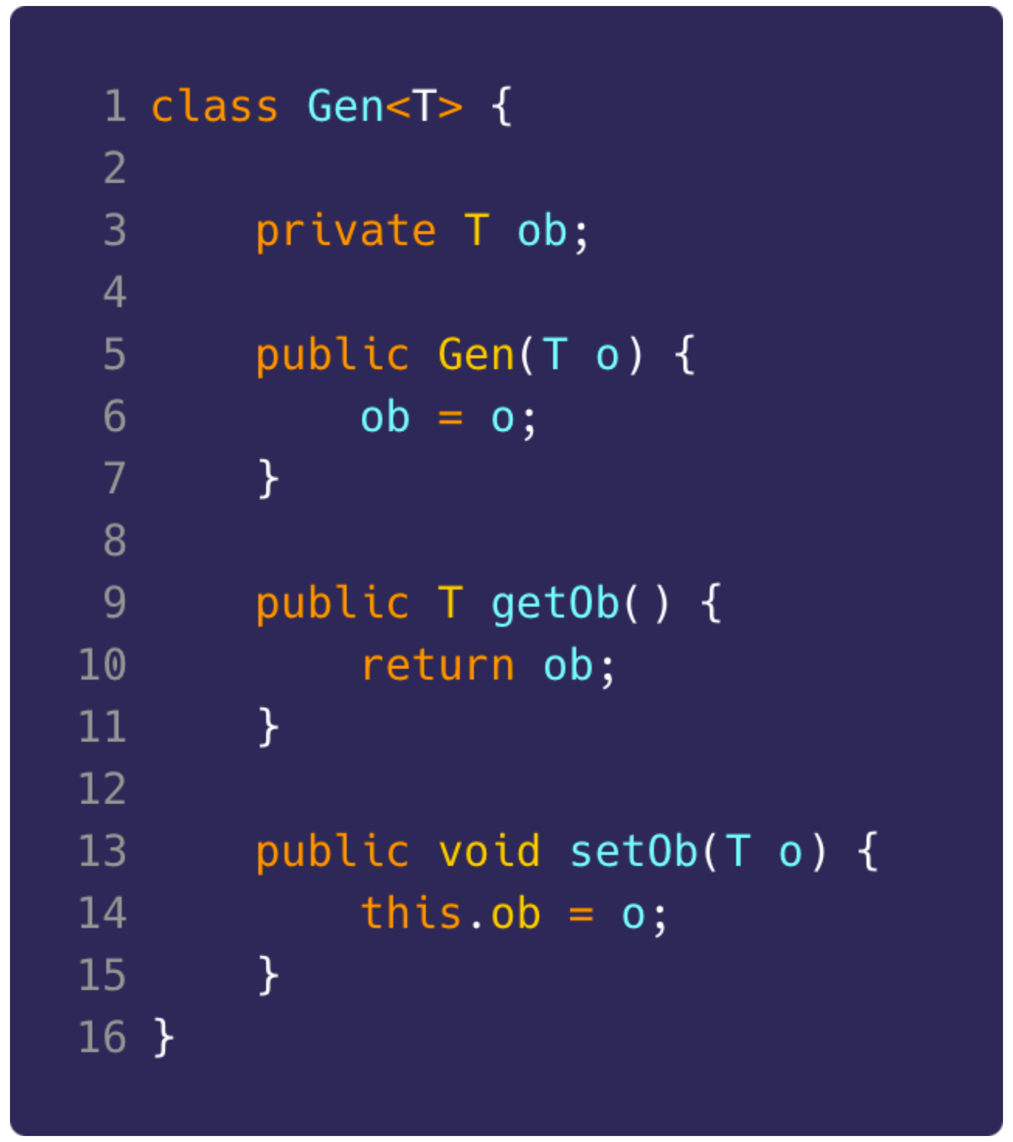
syntax: class class-name<type-param-list> {…}

type parameter scope: inside class definition block

- **type erasure:**

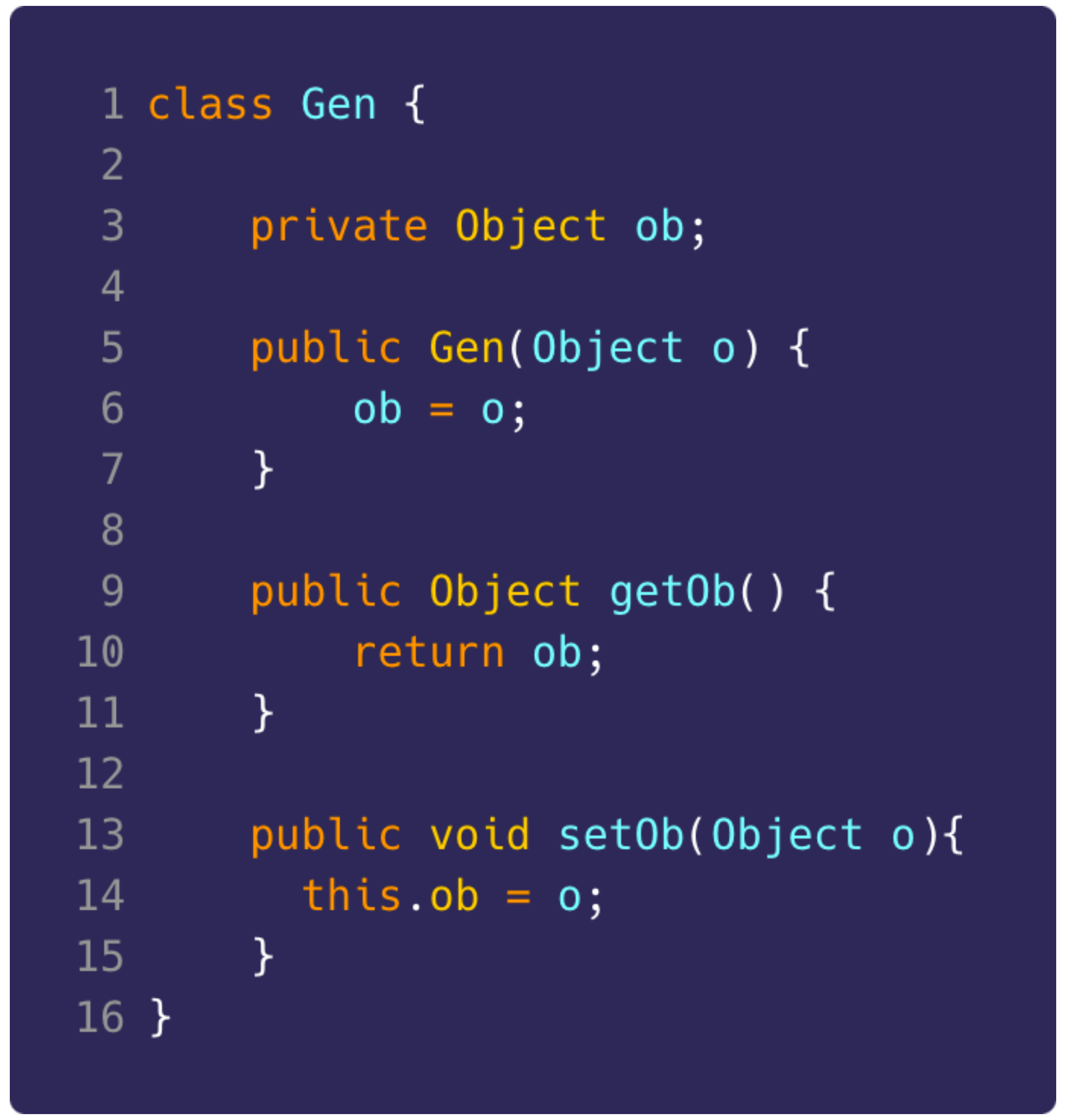
  To implements generics and to also preserve backward compatibility *javac* applies type erasure.

  - replaces all type parameters with their bounds or with `Object` if the parameter is unbounded
  - inserts type casts if necessary
  - Generates bridge methods to preserve polymorphism



```java
class Gen<T> {

    private T ob;

    public Gen(T o) {
        ob = o;
    }

    public T getOb() {
        return ob;
    }

    public void setOb(T o) {
        this.ob = o;
    }
}
```

type erasure



```java
class Gen {

    private Object ob;

    public Gen(Object o) {
        ob = o;
    }

    public Object getOb() {
        return ob;
    }

    public void setOb(Object o){
      this.ob = o;
    }
}
```

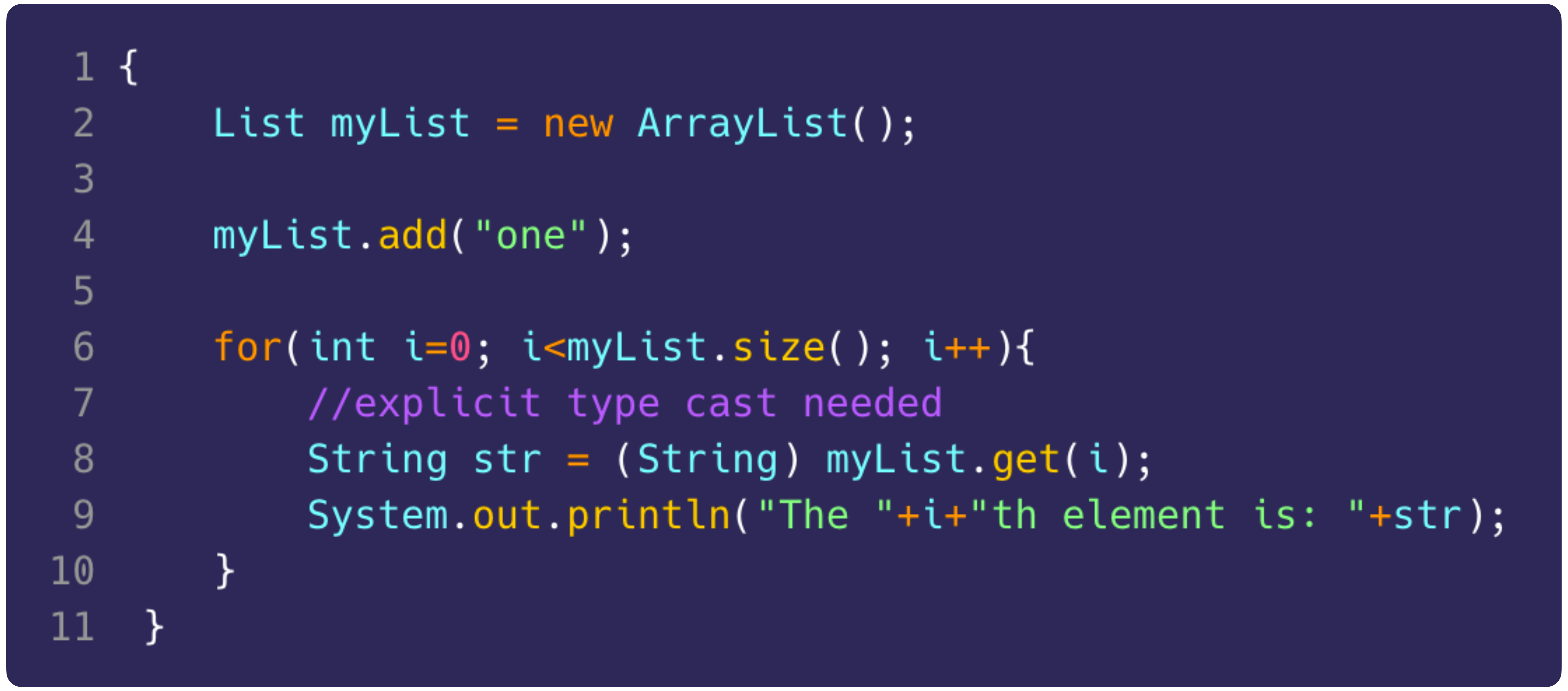
Raw types

- support for generics did not exist prior to *JDK 5*

- it was necessary to provide a transition path which enables pre-generics code to remain functional while at the same time being compatible with generics.

- to handle the transition to generics, Java allows a generic class to be used without any type arguments, resulting in ***raw type***

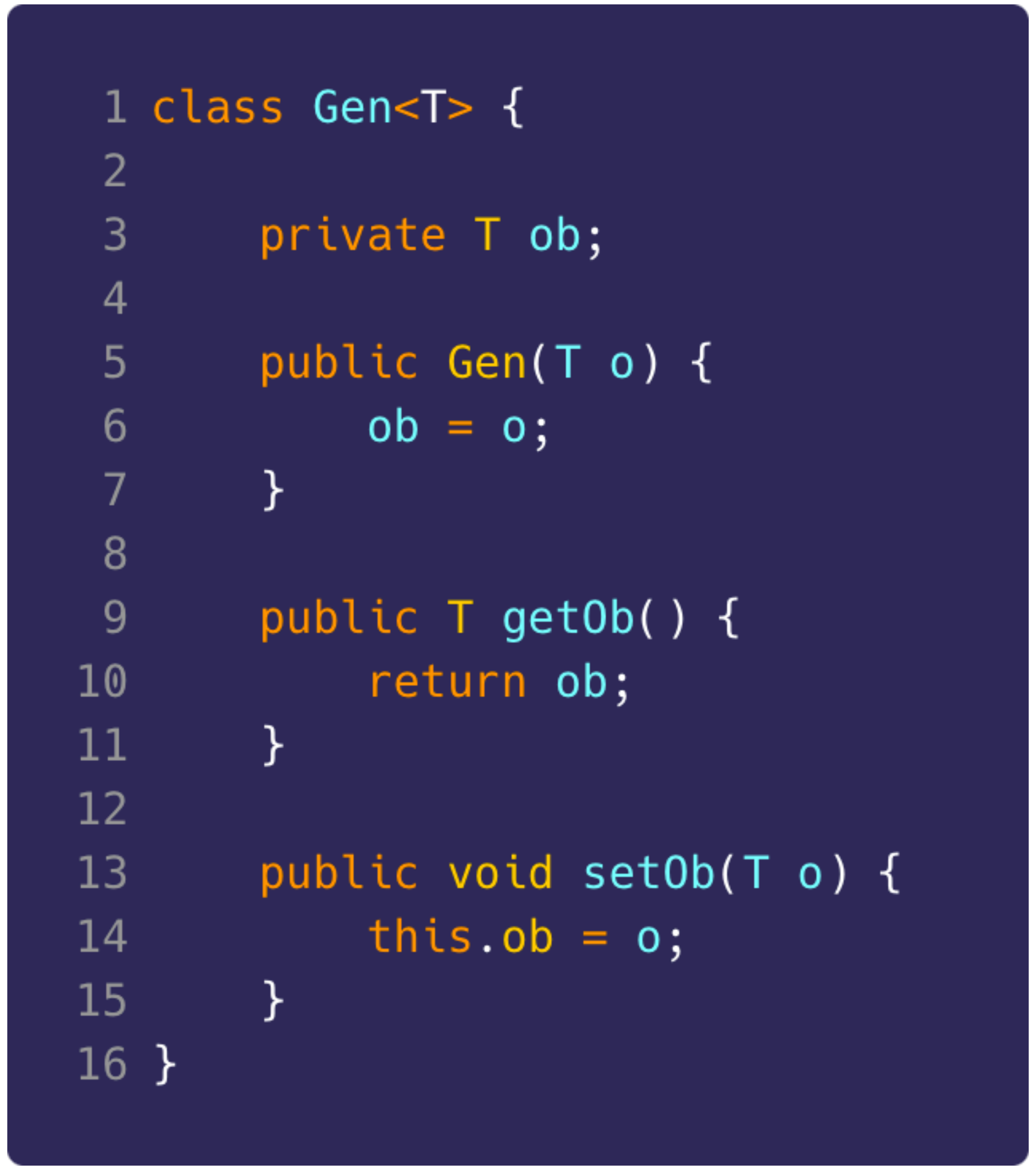
- **disadvantage:**
  type safety of generics is lost



```
{
    List myList = new ArrayList();

    myList.add("one");

    for(int i=0; i<myList.size(); i++){
        //explicit type cast needed
        String str = (String) myList.get(i);
        System.out.println("The "+i+"th element is: "+str);
    }
 }
```

Type parameter

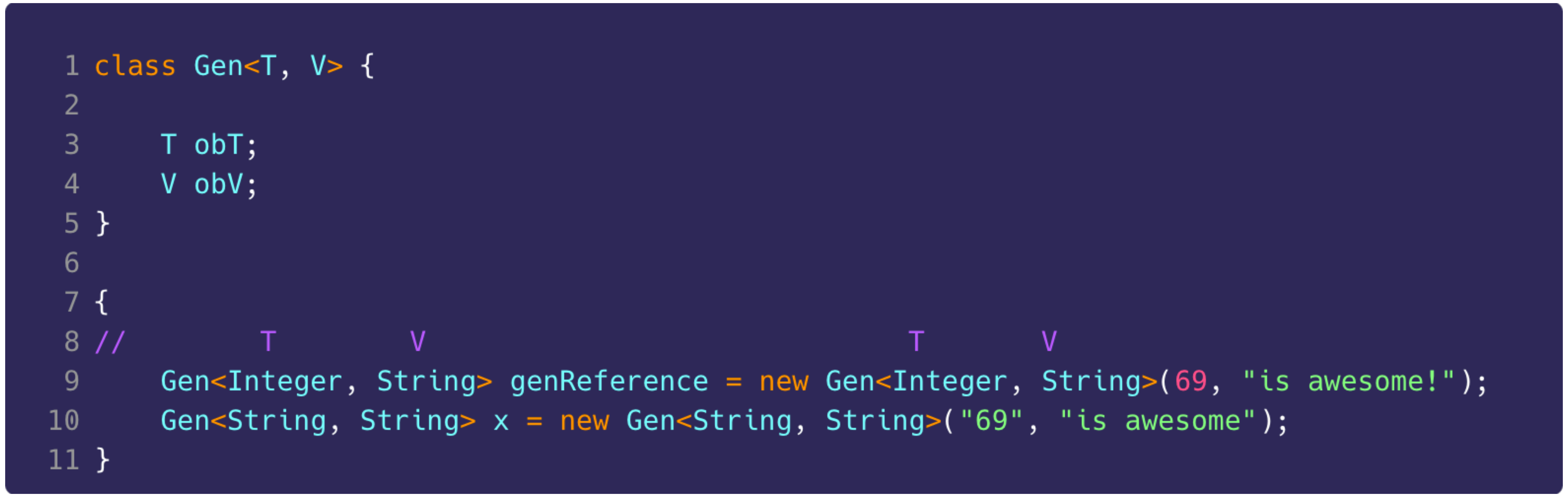
- scope depends on where it is declared
- needs to be declared inside <>
- no special significance to the name T
- naming conventions:
  - E: Element (mainly used by *Collection Framework*)
  - K: Key
  - N: Number
  - T: Type
  - V: Value
  - S, U, etc.



```java
class Gen<T> {

    private T ob;

    public Gen(T o) {
        ob = o;
    }

    public T getOb() {
        return ob;
    }

    public void setOb(T o) {
        this.ob = o;
    }
}
```

Type parameter

- *JDK 10*: cannot be ~~var~~
- arguments can only be reference types
  eg.: `Integer,List<Integer>, int[]`
- more than one type parameter in a comma-separated list



```java
class Gen<T, V> {

    T obT;
    V obV;
}

{
//        T        V                             T       V
    Gen<Integer, String> genReference = new Gen<Integer, String>(69, "is awesome!");
    Gen<String, String> x = new Gen<String, String>("69", "is awesome");
}
```

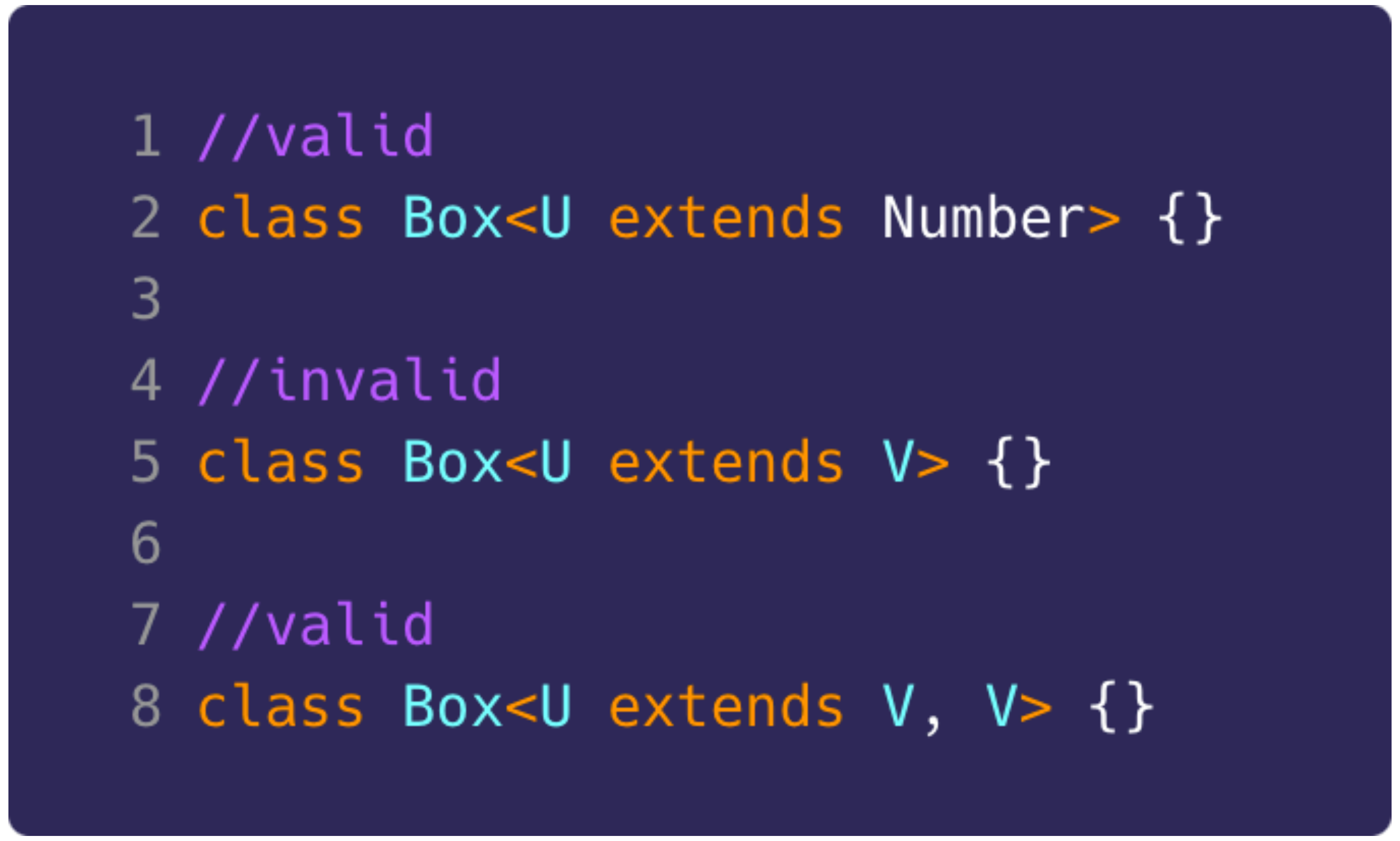
Type parameter - Bounding

We can use inheritance in order **to limit** the types that can be substituted.

```
syntax: <T extends superclass>
```

This specifies that T can only be replaced by superclass, or subclasses of superclass, thus superclass defines an inclusive, upper limit.

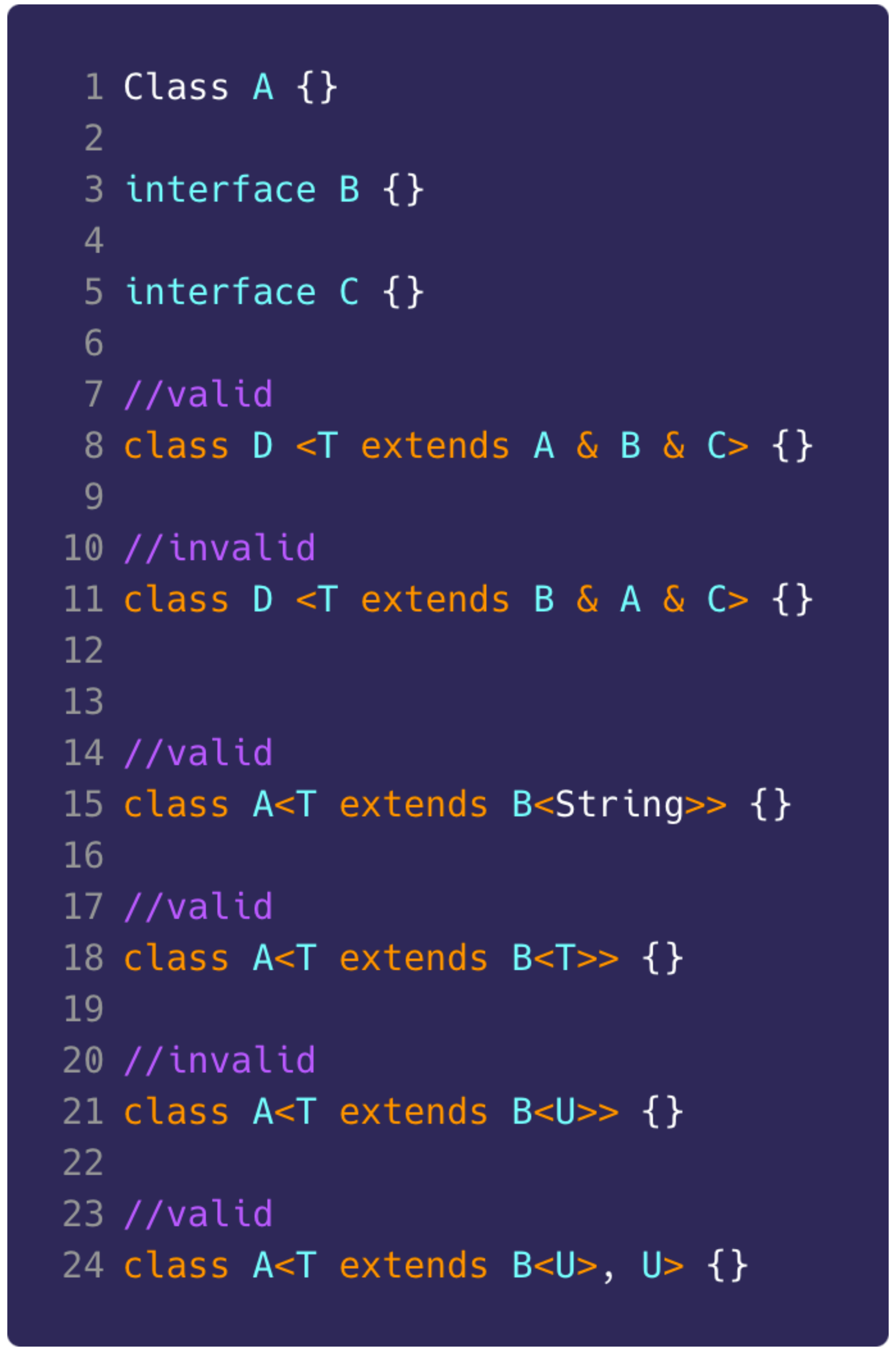
- limit can be an *interface* or a *class*
- only *upper* bound
- only specified types can be extended, else you have to specify it by listing it as a type parameter



```
//valid
class Box<U extends Number> {}

//invalid
class Box<U extends V> {}

//valid
class Box<U extends V, V> {}
```

Type parameter - Bounding

- a type parameter can have multiple bounds
  - & operator
  - only 1 class
  - class has to come first
- if the superclass is generic, and it doesn't have a specified argument, than it's type has to be listed in the parameter list

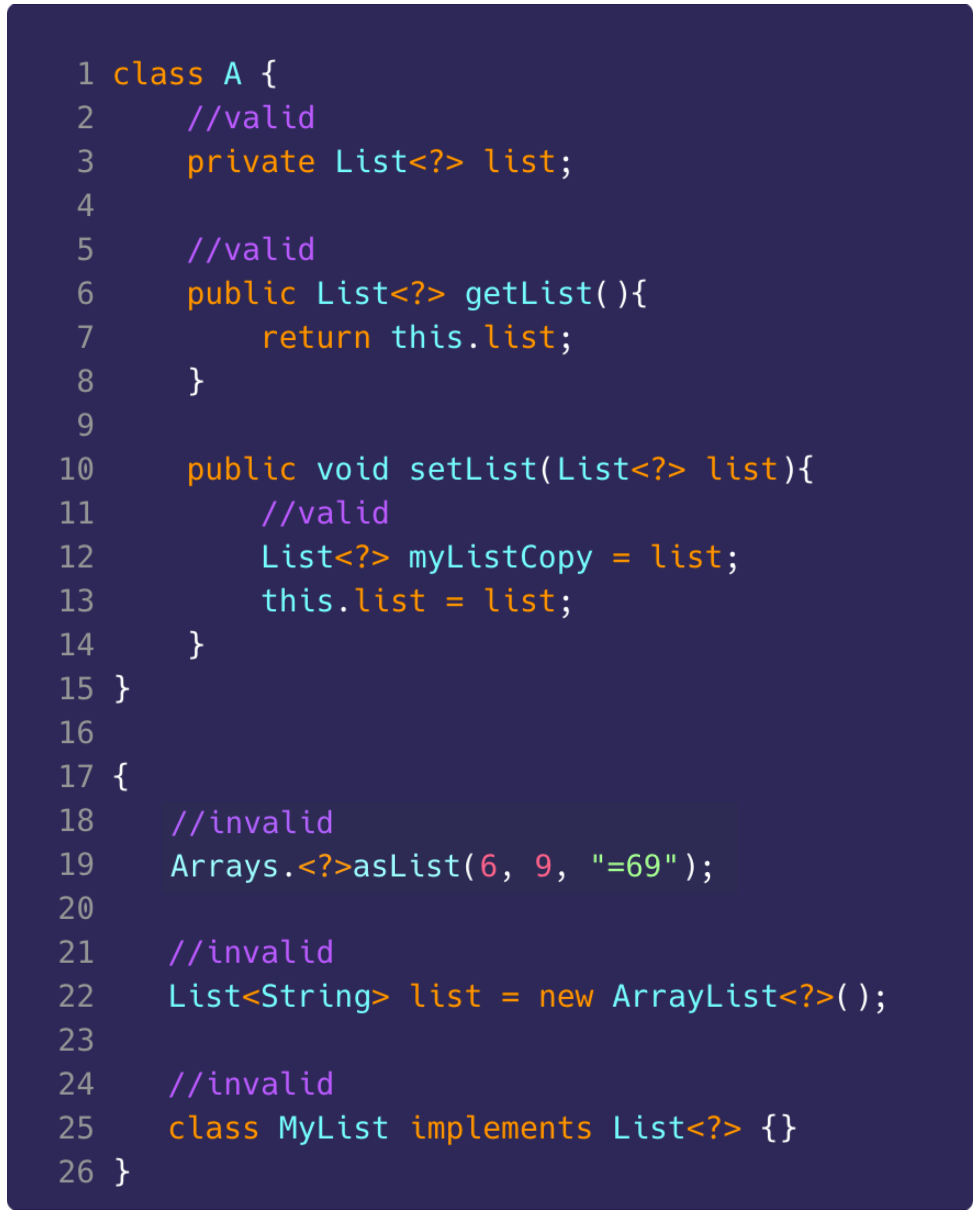


```
Class A {}

interface B {}

interface C {}

//valid
class D <T extends A & B & C> {}

//invalid
class D <T extends B & A & C> {}


//valid
class A<T extends B<String>> {}

//valid
class A<T extends B<T>> {}

//invalid
class A<T extends B<U>> {}

//valid
class A<T extends B<U>, U> {}
```

Wildcard argument

In generic code, the question mark (**?**), called the **wildcard**, represents an **unknown type**.

- **can** use a generic type with a wildcard:
  - as a type of a parameter
  - as a field
  - as a local variable
- **can’t** use a generic type with a wildcard:
  - as a type argument for a generic method invocation
  - at generic class instance creation
  - as a supertype

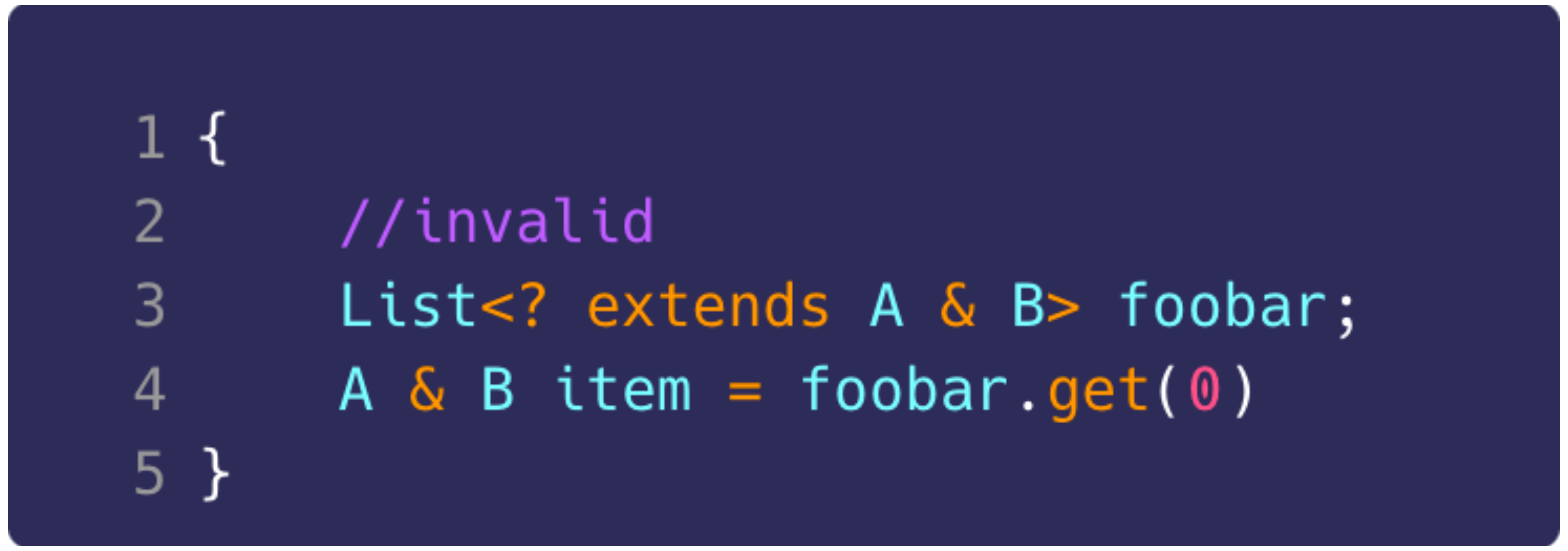


```java
class A {
    //valid
    private List<?> list;

    //valid
    public List<?> getList(){
        return this.list;
    }

    public void setList(List<?> list){
        //valid
        List<?> myListCopy = list;
        this.list = list;
    }
}

{
    //invalid
    Arrays.<?>asList(6, 9, "=69");

    //invalid
    List<String> list = new ArrayList<?>();

    //invalid
    class MyList implements List<?> {}
}
```

Wildcard argument - Bounding

We can use inheritance in order **to limit** the types that can be substituted.

- limit can be an *interface* or a *class*
- works only with specified types
- no multiple bounds
- upper bound ***and*** lower bound

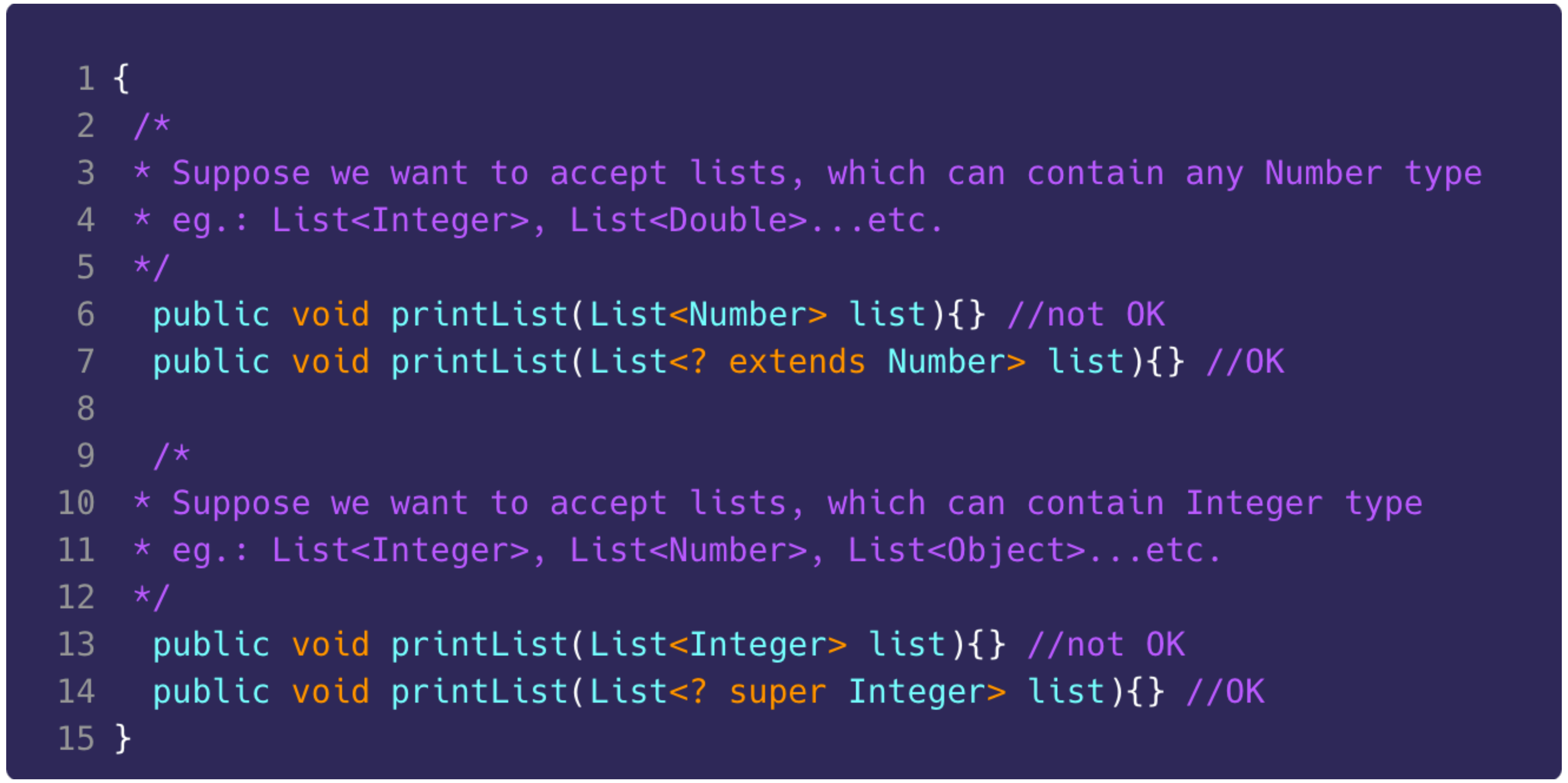


```
{
    //invalid
    List<? extends A & B> foobar;
    A & B item = foobar.get(0)
}
```

Wildcard argument - Bounding

- *Upper:* `syntax: <? extends superclass>`
  This specifies, that `?` can only be superclass type, or a type that is a subclass of the superclass, thus superclass defines an inclusive, upper limit.

- Lower: `syntax: <? super subclass>`
  This specifies, that `?` can only be subclass type, or a type that is a superclass of the subclass, thus subclass defines an inclusive, lower limit.



```
{
 /*
 * Suppose we want to accept lists, which can contain any Number type
 * eg.: List<Integer>, List<Double>...etc.
 */
  public void printList(List<Number> list){} //not OK
  public void printList(List<? extends Number> list){} //OK

  /*
 * Suppose we want to accept lists, which can contain Integer type
 * eg.: List<Integer>, List<Number>, List<Object>...etc.
 */
  public void printList(List<Integer> list){} //not OK
  public void printList(List<? super Integer> list){} //OK
}
```

2. Method

A method is generic if it declares one or more type variables.
These type variables are known as the formal type parameters of the method.

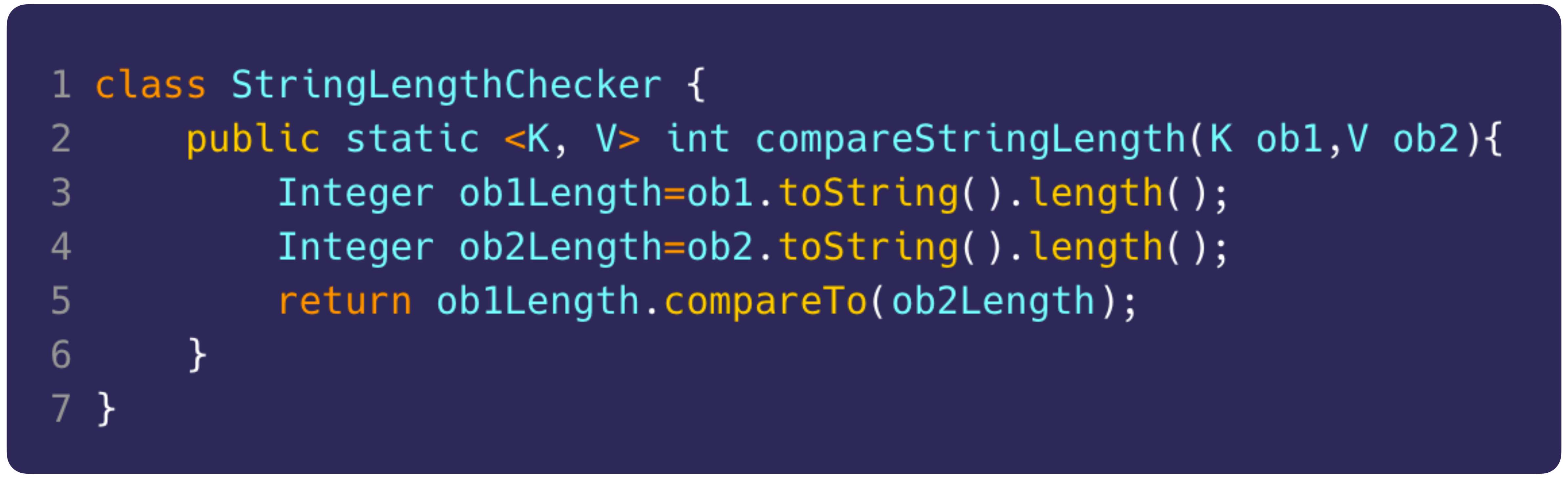
/Generic methods are methods that introduce their own type parameters./

syntax: <formal-type-param-list > ret-type meth-name (param-list) {…}

formal type parameter scope: inside method definition block

- same type parameter rules apply
- can be declared in generic and in non-generic class or interface
- you can explicitly specify the type argument if needed on calling:

```
StringLengthChecker.
<Integer, String>compareStringLength(69, ”ok”)
```



```
class StringLengthChecker {
    public static <K, V> int compareStringLength(K ob1,V ob2){
        Integer ob1Length=ob1.toString().length();
        Integer ob2Length=ob2.toString().length();
        return ob1Length.compareTo(ob2Length);
    }
}
```

3. Constructor

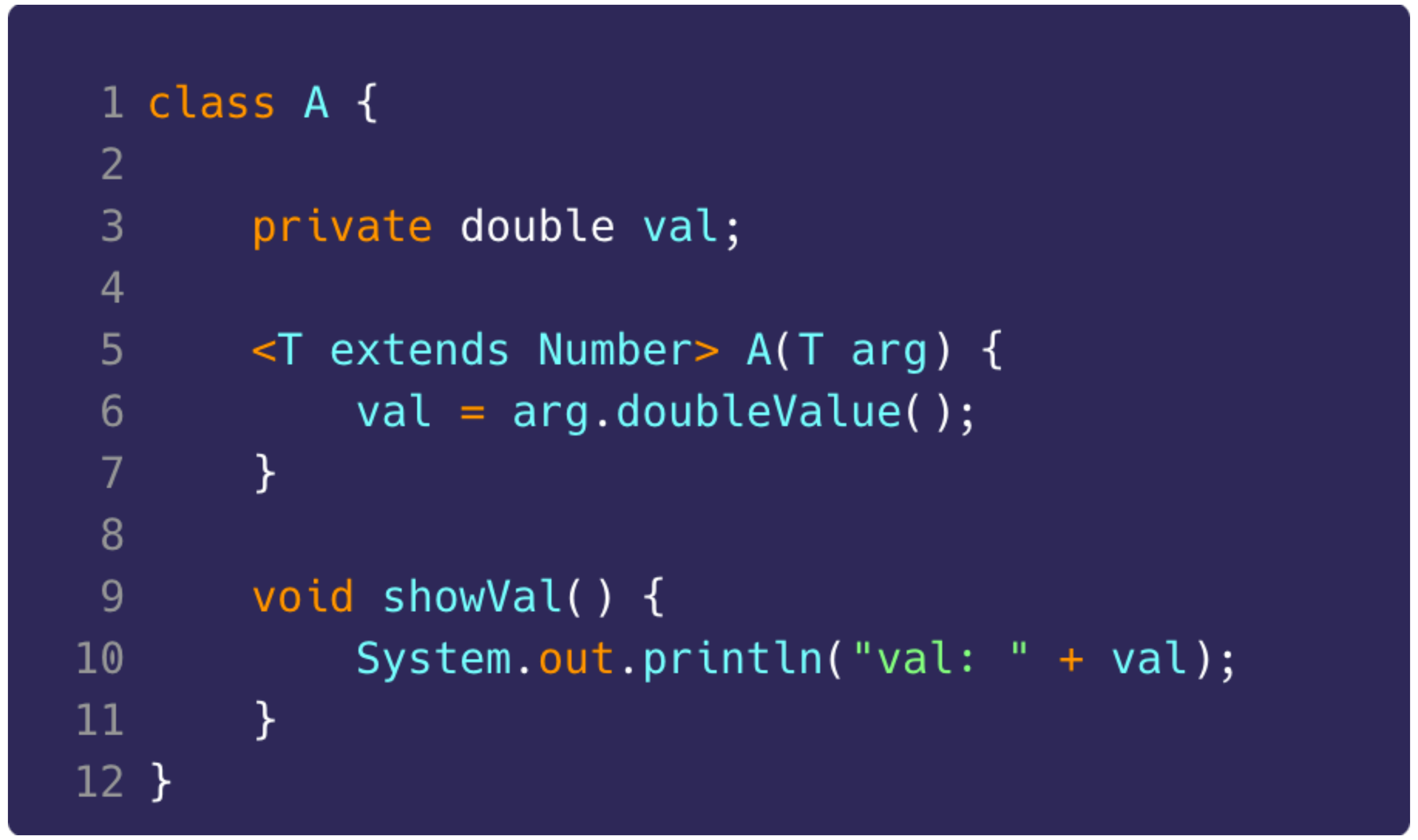
```
A constructor is generic if it declares one or more type variables.
These type variables are known as the formal type parameters of the constructor.
```

```
/Generic constructors are constructors that introduce their own type parameters./
```

```
syntax: <formal-type-param-list > constructor-name (param-list) {…}
```

```
formal type parameter scope: inside constructor definition block
```

- same type parameter rules apply
- can be declared in generic and in non-generic class
- you cannot explicitly specify the type argument on calling

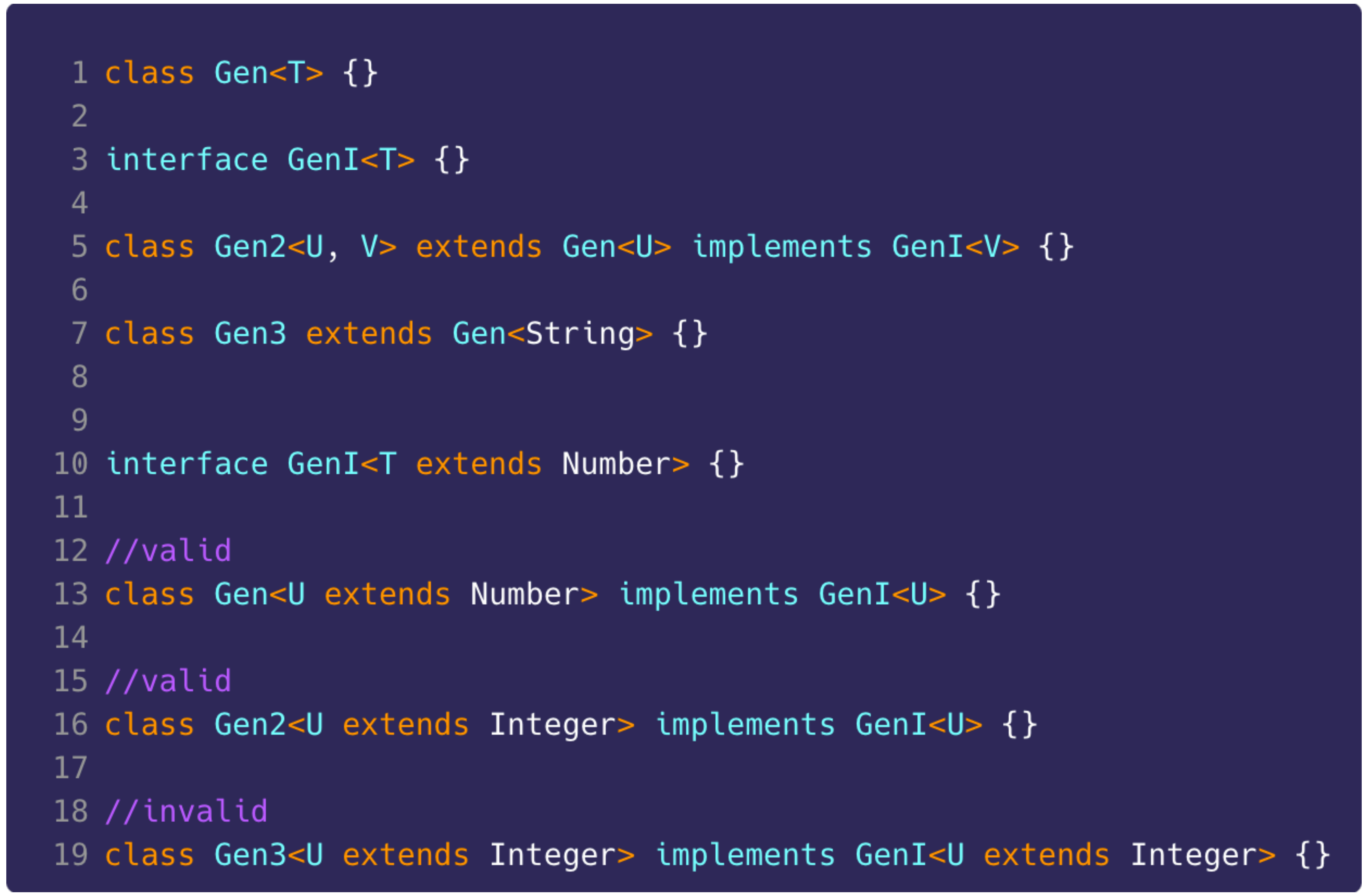


```
class A {

    private double val;

    <T extends Number> A(T arg) {
        val = arg.doubleValue();
    }

    void showVal() {
        System.out.println("val: " + val);
    }
}
```

Inheritance

In a generic hierarchy, any type arguments needed by a generic superclass or interface must be passed up the hierarchy by all subclasses.

- only the not specified type arguments have to be passed!
- if the needed type argument is a bounded type, then the inheritor has to define its bound.
  Once this bound has been established, its not allowed to specify it again!



```java
class Gen<T> {}

interface GenI<T> {}

class Gen2<U, V> extends Gen<U> implements GenI<V> {}

class Gen3 extends Gen<String> {}


interface GenI<T extends Number> {}

//valid
class Gen<U extends Number> implements GenI<U> {}

//valid
class Gen2<U extends Integer> implements GenI<U> {}

//invalid
class Gen3<U extends Integer> implements GenI<U extends Integer> {}
```

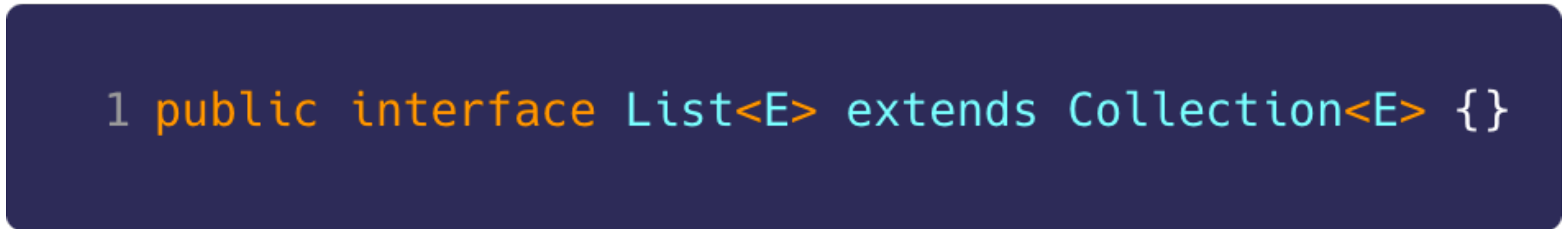
4. Interface

An interface is generic if it declares one or more type variables. These type variables are known as the type parameters of the interface.

syntax: interface interface-name<type-param-list> {…}

type parameter scope: inside interface definition block

- same type parameter rules apply

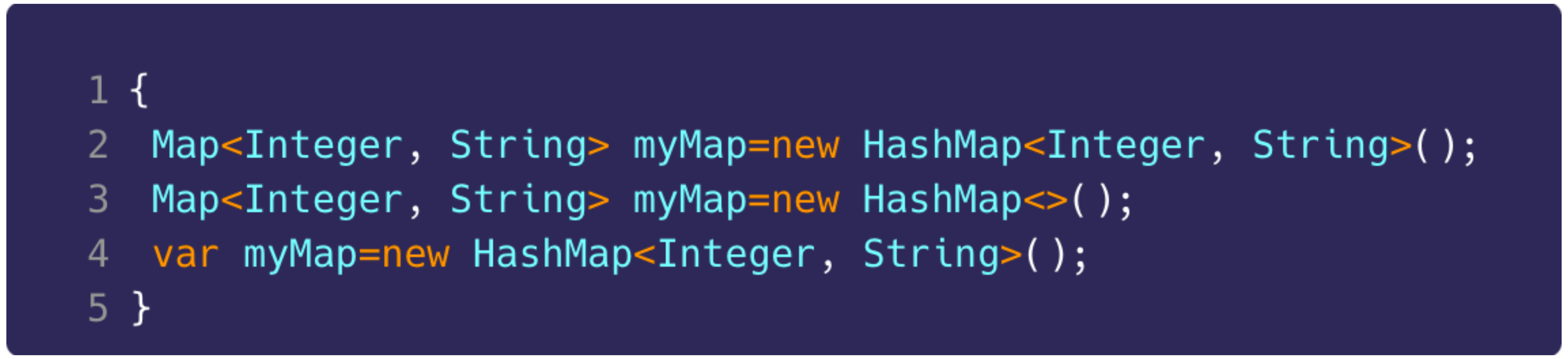


```java
public interface List<E> extends Collection<E> {}
```

Type inference

`Type inference is the ability of the Java compiler to infer datatypes based on corresponding declarations.`

- *JDK 7* introduced the diamond operator `<>`
    - prior to JDK 7: `class-name<type-arg-list> var-name = new class-name <same-type-arg-list>(cons-arg-list);`
    - after JDK 7: `class-name<type-arg-list > var-name = new class-name <>(cons-arg-list);`
- *JDK 10* introduced the `var` keyword
    - after JDK 10: `var var-name = new class-name <type-arg-list>(cons-arg-list);`

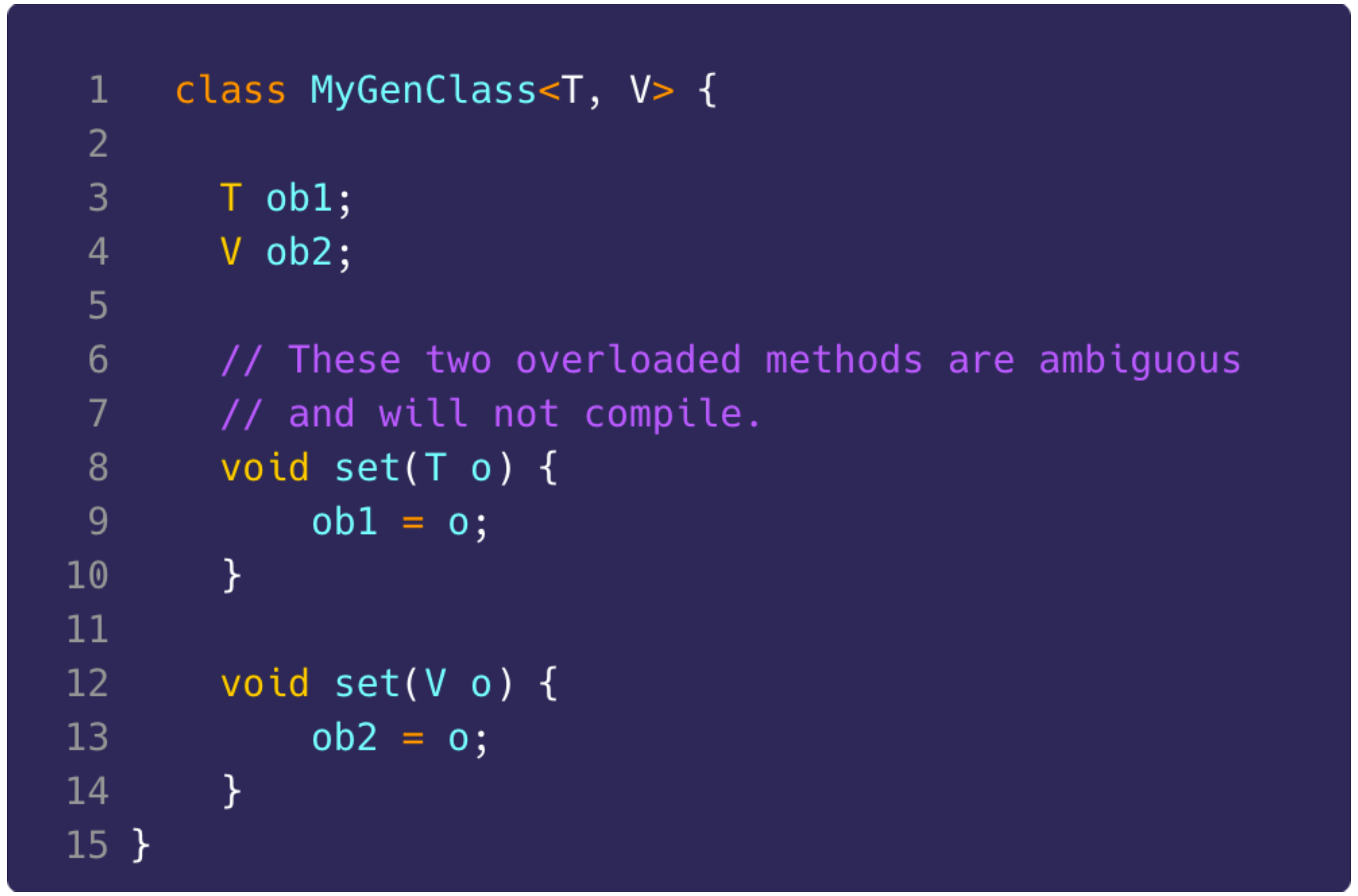


```
{
 Map<Integer, String> myMap=new HashMap<Integer, String>();
 Map<Integer, String> myMap=new HashMap<>();
 var myMap=new HashMap<Integer, String>();
}
```

Ambiguity error

```
Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same
erased type, causing a conflict, resulting a compile time error.
```

- mainly happens on **overloading**
  (same name, same class or interface, different signature)

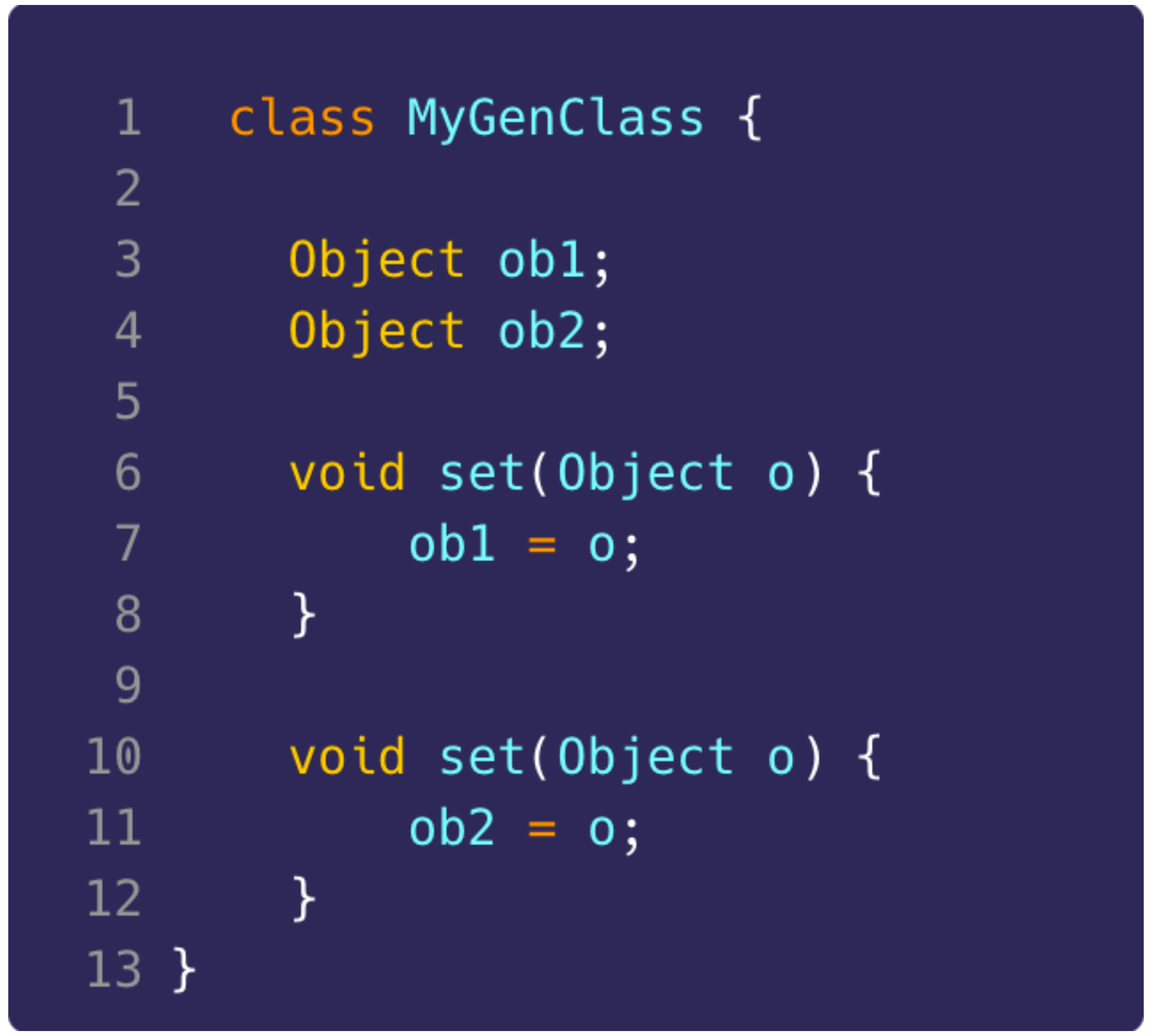


```java
class MyGenClass<T, V> {

  T ob1;
  V ob2;

  // These two overloaded methods are ambiguous
  // and will not compile.
  void set(T o) {
      ob1 = o;
  }

  void set(V o) {
      ob2 = o;
  }
}
```

type erasure

↔

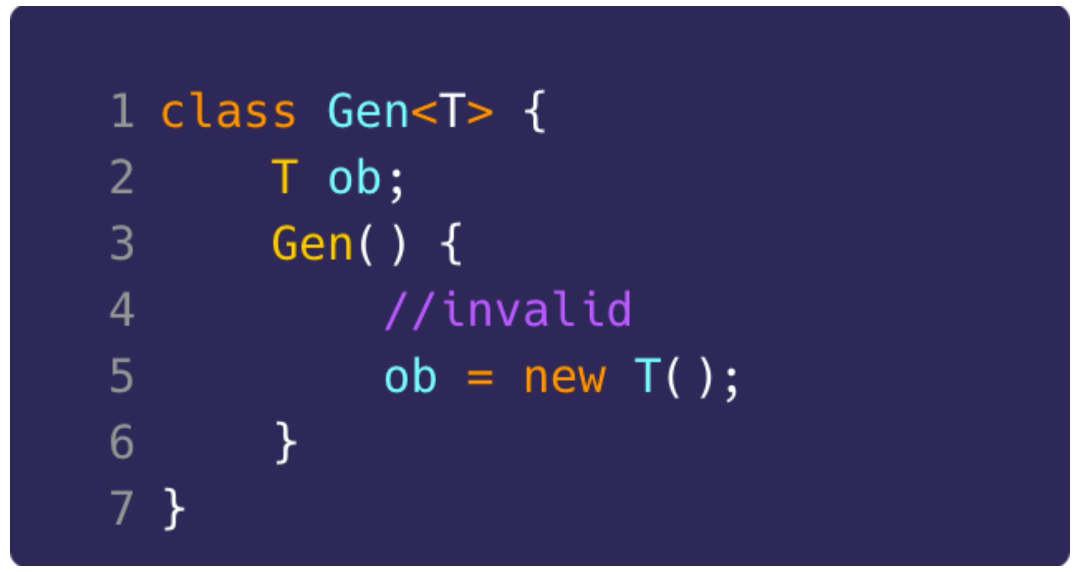
= compile-time error!



```java
class MyGenClass {

  Object ob1;
  Object ob2;

  void set(Object o) {
      ob1 = o;
  }

  void set(Object o) {
      ob2 = o;
  }
}
```

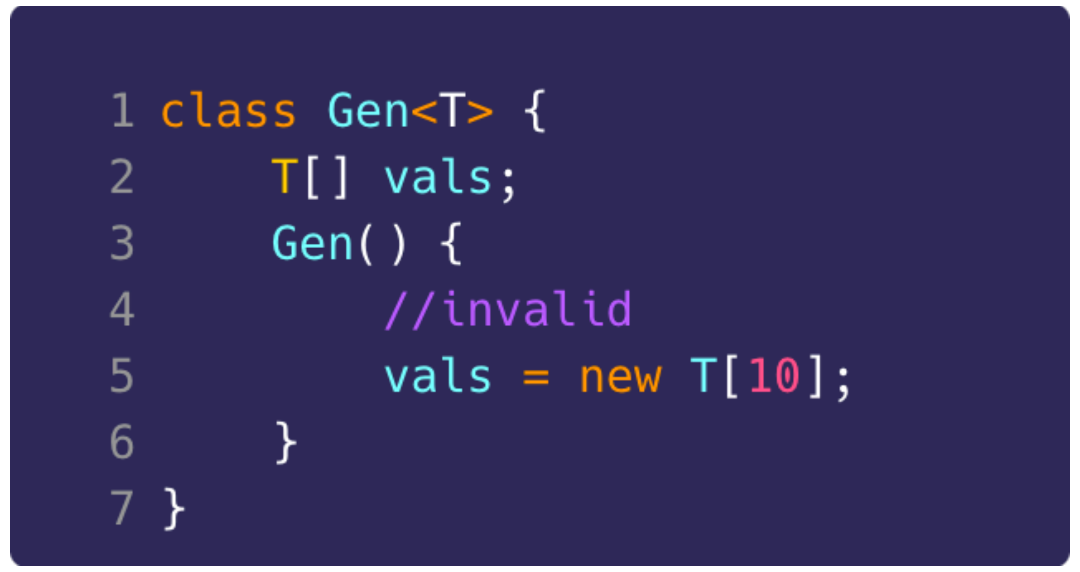
Some restrictions

- cannot instantiate a **type parameter**



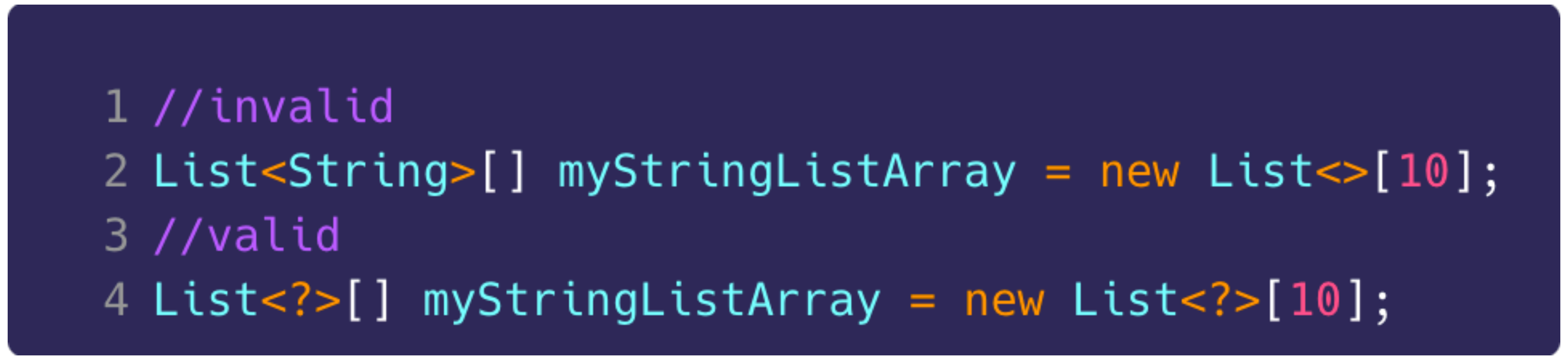
```java
class Gen<T> {
    T ob;
    Gen() {
        //invalid
        ob = new T();
    }
}
```

- cannot instantiate an **array** whose element type is a *type parameter*



```java
class Gen<T> {
    T[] vals;
    Gen() {
        //invalid
        vals = new T[10];
    }
}
```

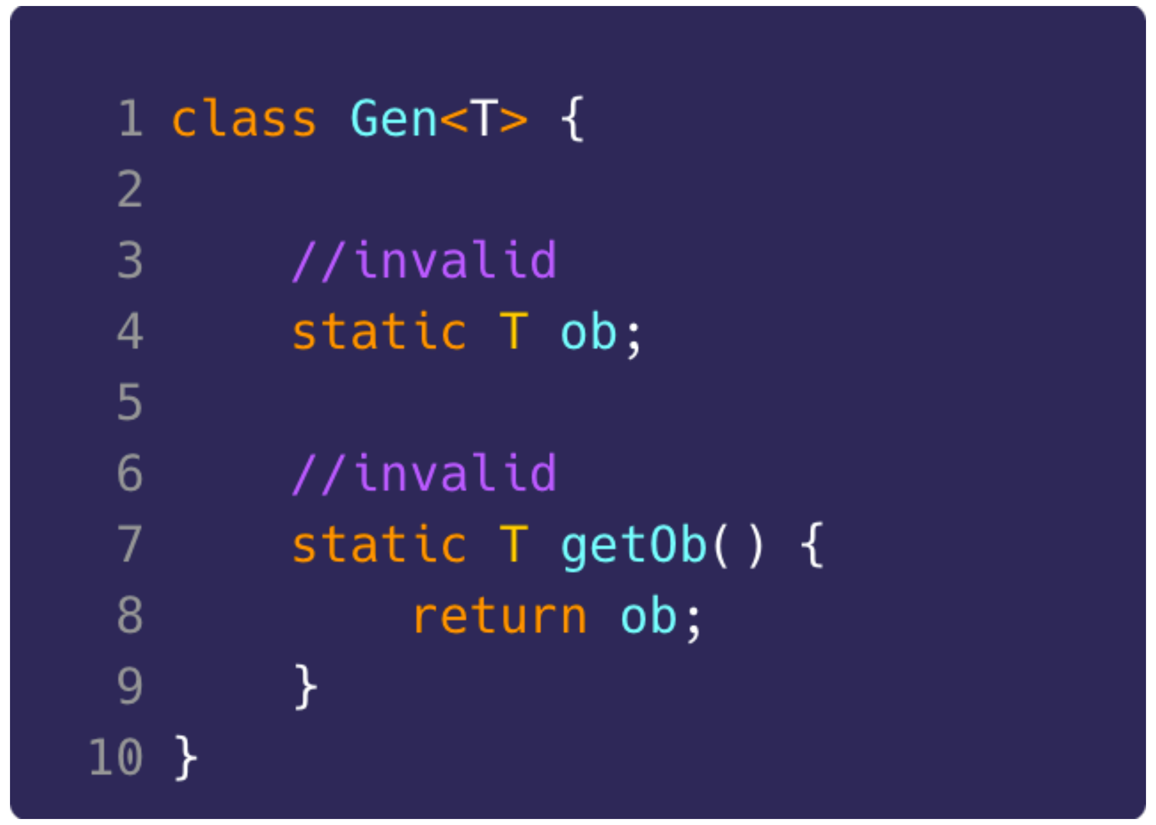
- cannot create an **array** of *type-specific generic references*



```java
//invalid
List<String>[] myStringListArray = new List<>[10];
//valid
List<?>[] myStringListArray = new List<?>[10];
```

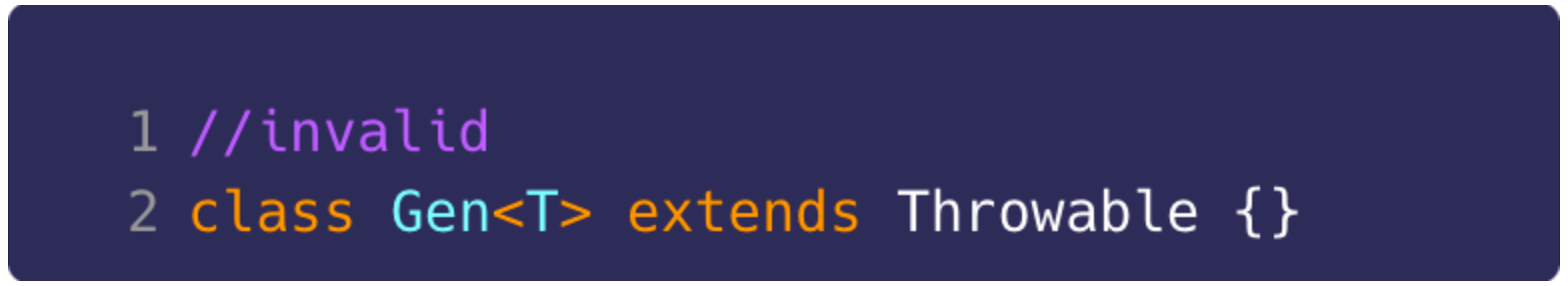
Some restrictions

- **no static member** can use a *type parameter* declared by the enclosing class



```java
class Gen<T> {

    //invalid
    static T ob;

    //invalid
    static T getOb() {
        return ob;
    }
}
```

- a generic class cannot extend **Throwable**, meaning you cannot create generic exception classes



```java
//invalid
class Gen<T> extends Throwable {}
```

Generics

Summary

introduced by JDK 5, no explicit type-casting, type check at compile-time, can be class | interface | method | constructor, type erasure, raw types, type parameter and it’s rules, type parameter bounding, wildcard argument (`?`), wildcard argument and it’s bounding, inheritance between generics, type inference (`<>` JDK7, `var` JDK10), ambiguity compile time error, some restrictions

- *further reading: bridge methods*

The End