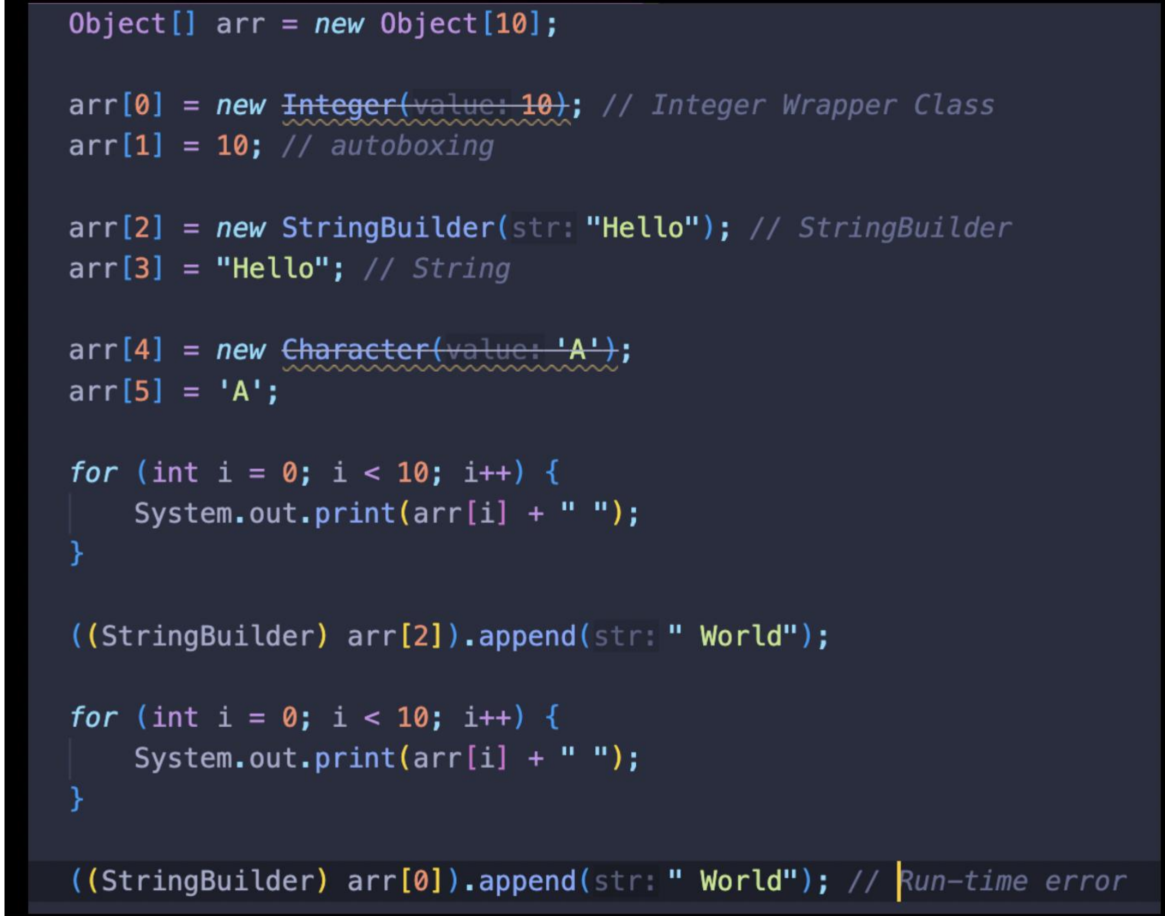
Generics in Java

↳ Parameterized Types
- → Reusability & Generalizatn
- → Type Safety (no need of typecasting)

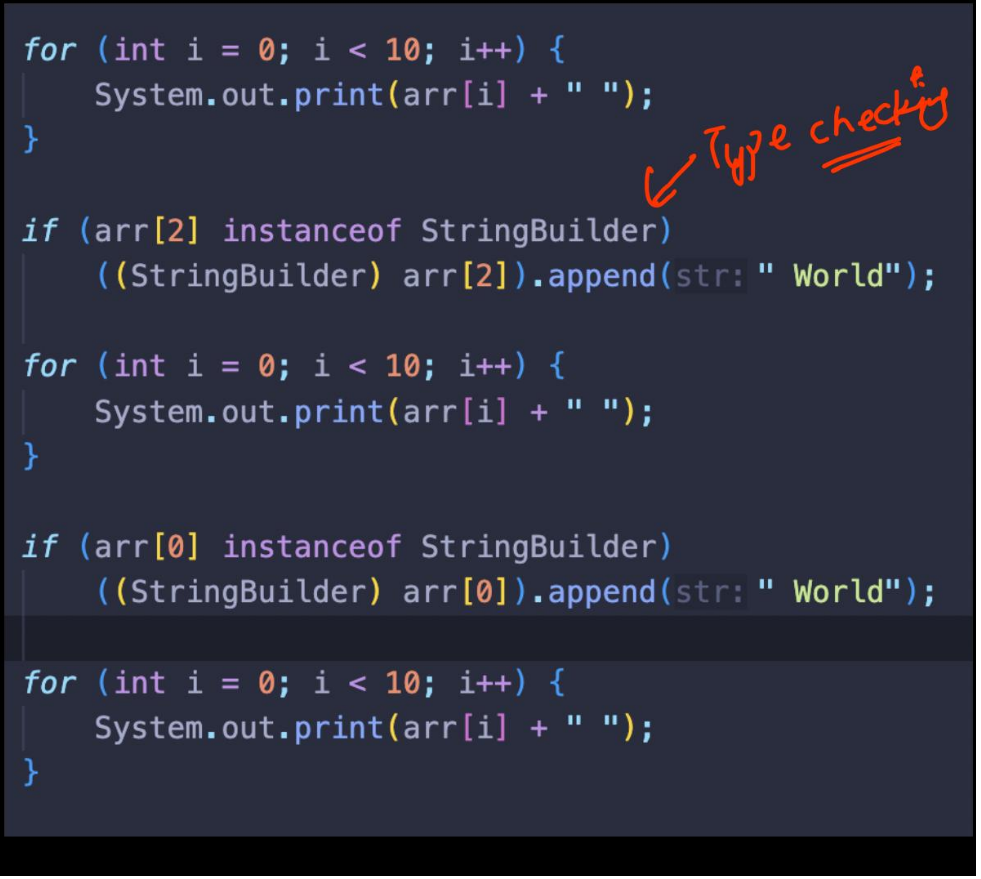
① Inbuilt Arraylist
- ↳ of Generic Type (Object type)
- ↳ of Specific Type

② Custom Arraylist → Q) why collectn framework of generics is not possible?
- ↳ of Integer Type
- ↳ of Generic Type (using Object)

Generics using Object class

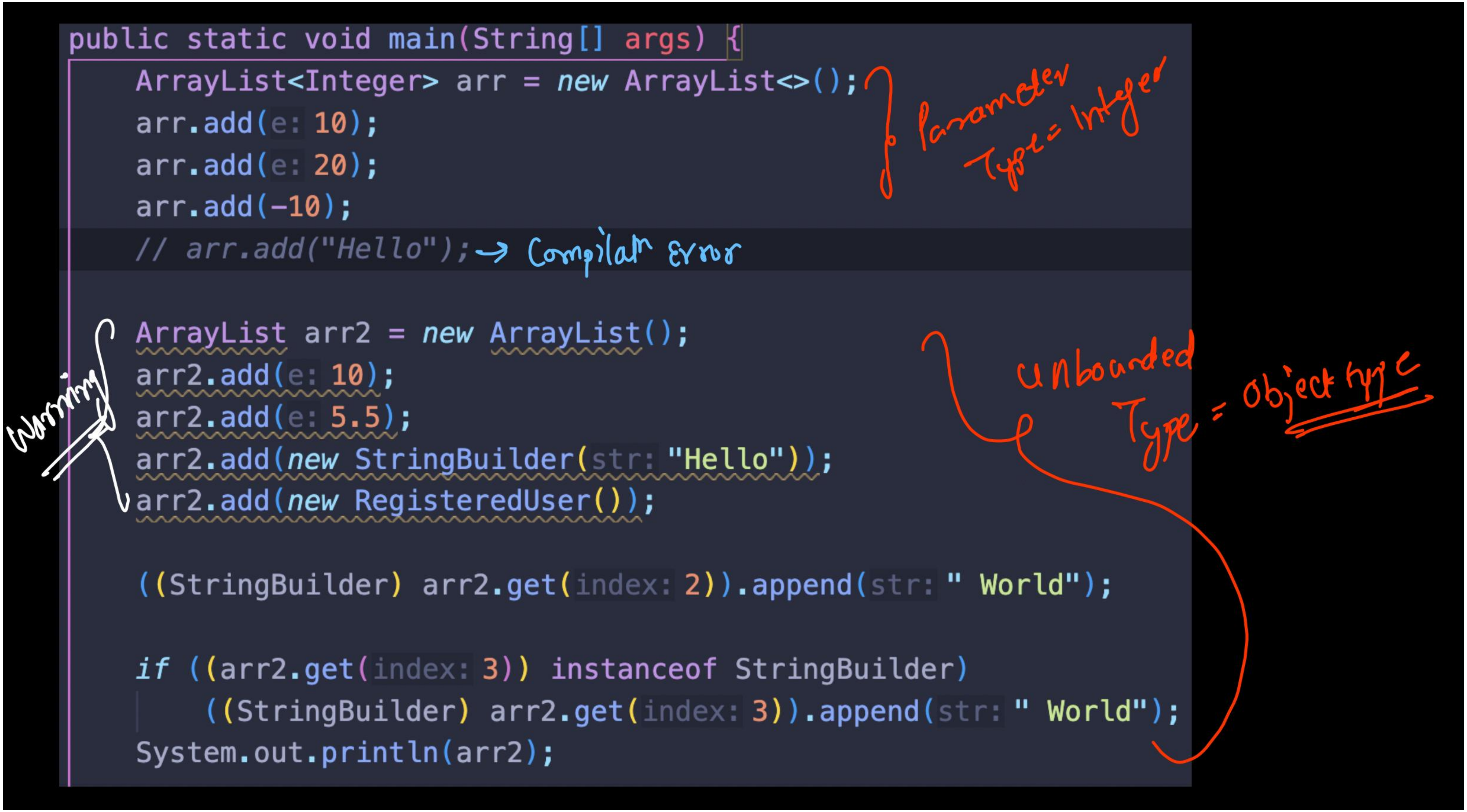


```java
Object[] arr = new Object[10];

arr[0] = new ~~Integer(value: 10)~~; // Integer Wrapper Class
arr[1] = 10; // autoboxing

arr[2] = new StringBuilder(str: "Hello"); // StringBuilder
arr[3] = "Hello"; // String

arr[4] = new ~~Character(value: 'A')~~;
arr[5] = 'A';

for (int i = 0; i < 10; i++) {
    System.out.print(arr[i] + " ");
}

((StringBuilder) arr[2]).append(str: " World");

for (int i = 0; i < 10; i++) {
    System.out.print(arr[i] + " ");
}

((StringBuilder) arr[0]).append(str: " World"); // Run-time error
```



Type checking

```java
for (int i = 0; i < 10; i++) {
    System.out.print(arr[i] + " ");
}

if (arr[2] instanceof StringBuilder)
    ((StringBuilder) arr[2]).append(str: " World");

for (int i = 0; i < 10; i++) {
    System.out.print(arr[i] + " ");
}

if (arr[0] instanceof StringBuilder)
    ((StringBuilder) arr[0]).append(str: " World");

for (int i = 0; i < 10; i++) {
    System.out.print(arr[i] + " ");
}
```



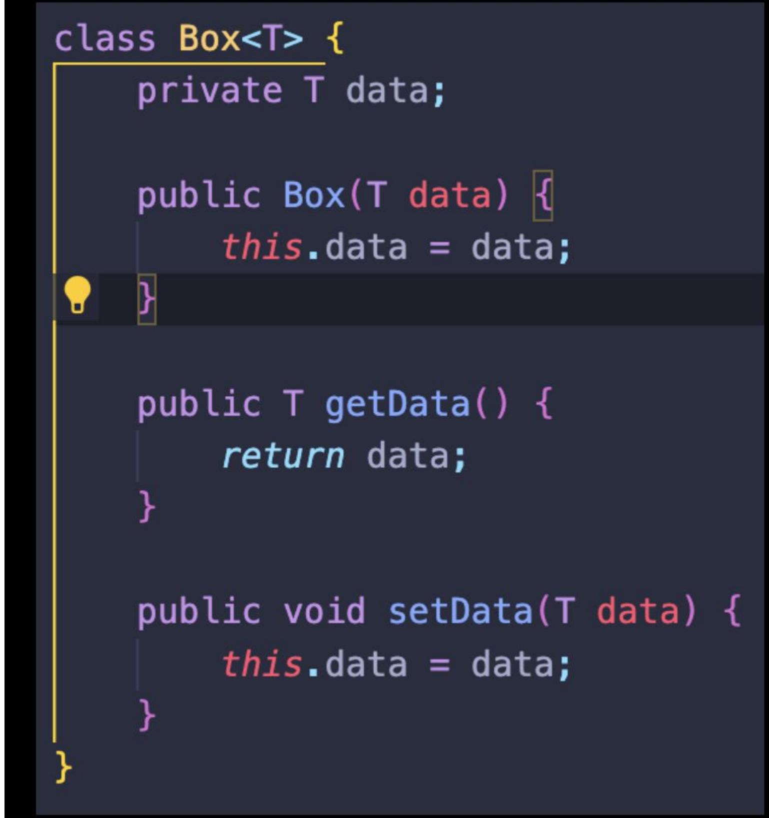
```java
public static void main(String[] args) {
    ArrayList<Integer> arr = new ArrayList<>();
    arr.add(e: 10);
    arr.add(e: 20);
    arr.add(-10);
    // arr.add("Hello");

    ArrayList arr2 = new ArrayList();
    arr2.add(e: 10);
    arr2.add(e: 5.5);
    arr2.add(new StringBuilder(str: "Hello"));
    arr2.add(new RegisteredUser());

    ((StringBuilder) arr2.get(index: 2)).append(str: " World");

    if ((arr2.get(index: 3)) instanceof StringBuilder)
        ((StringBuilder) arr2.get(index: 3)).append(str: " World");
    System.out.println(arr2);
```

Parameter Type = Integer

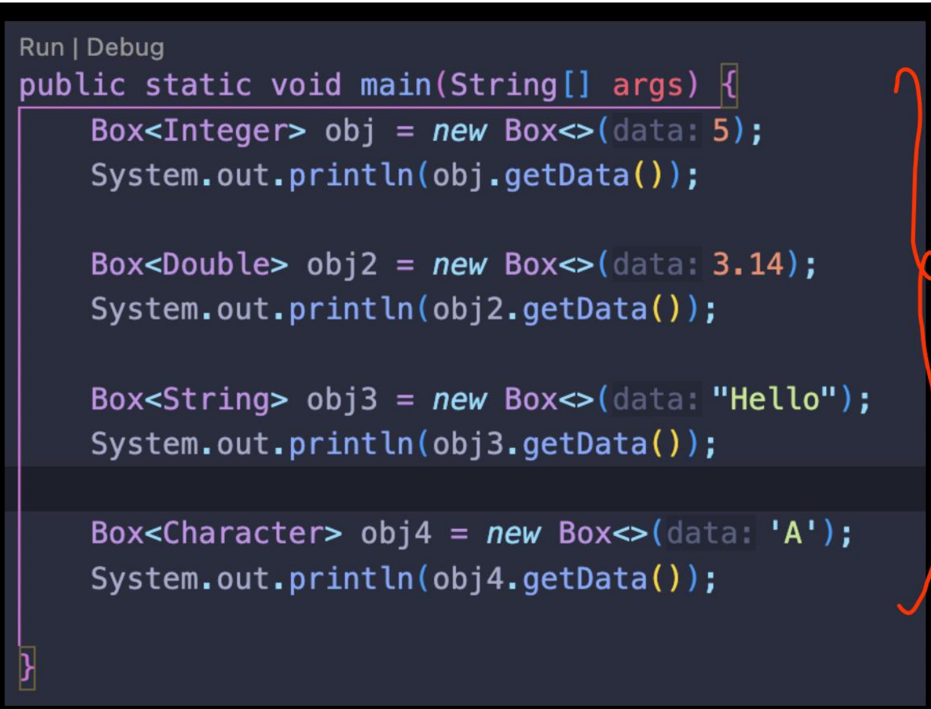
→ Compilatn Error

Warning

Unbounded Type = Object type



```java
class Box<T> {
    private T data;

    public Box(T data) {
        this.data = data;
    }

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }
}
```

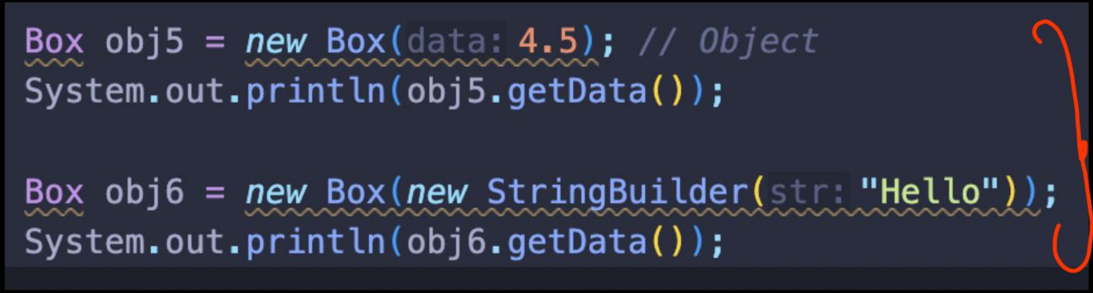


Run | Debug

```java
public static void main(String[] args) {
    Box<Integer> obj = new Box<>(data: 5);
    System.out.println(obj.getData());

    Box<Double> obj2 = new Box<>(data: 3.14);
    System.out.println(obj2.getData());

    Box<String> obj3 = new Box<>(data: "Hello");
    System.out.println(obj3.getData());

    Box<Character> obj4 = new Box<>(data: 'A');
    System.out.println(obj4.getData());

}
```

bounded

T will be replaced by parameter type.

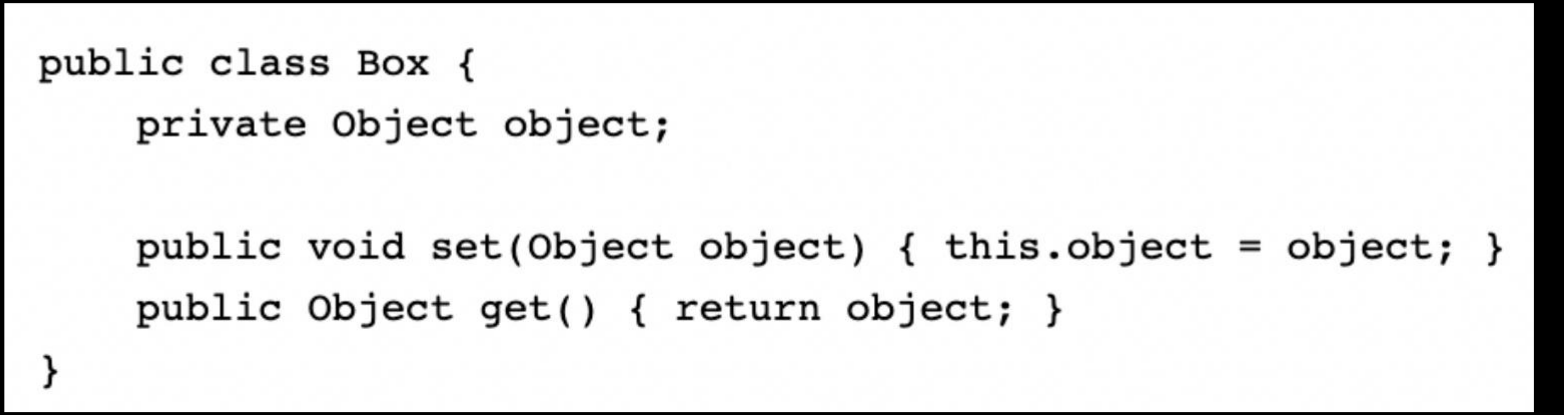


```java
Box obj5 = new Box(data: 4.5); // Object
System.out.println(obj5.getData());

Box obj6 = new Box(new StringBuilder(str: "Hello"));
System.out.println(obj6.getData());
```

unbounded
↓
T is replaced by Object

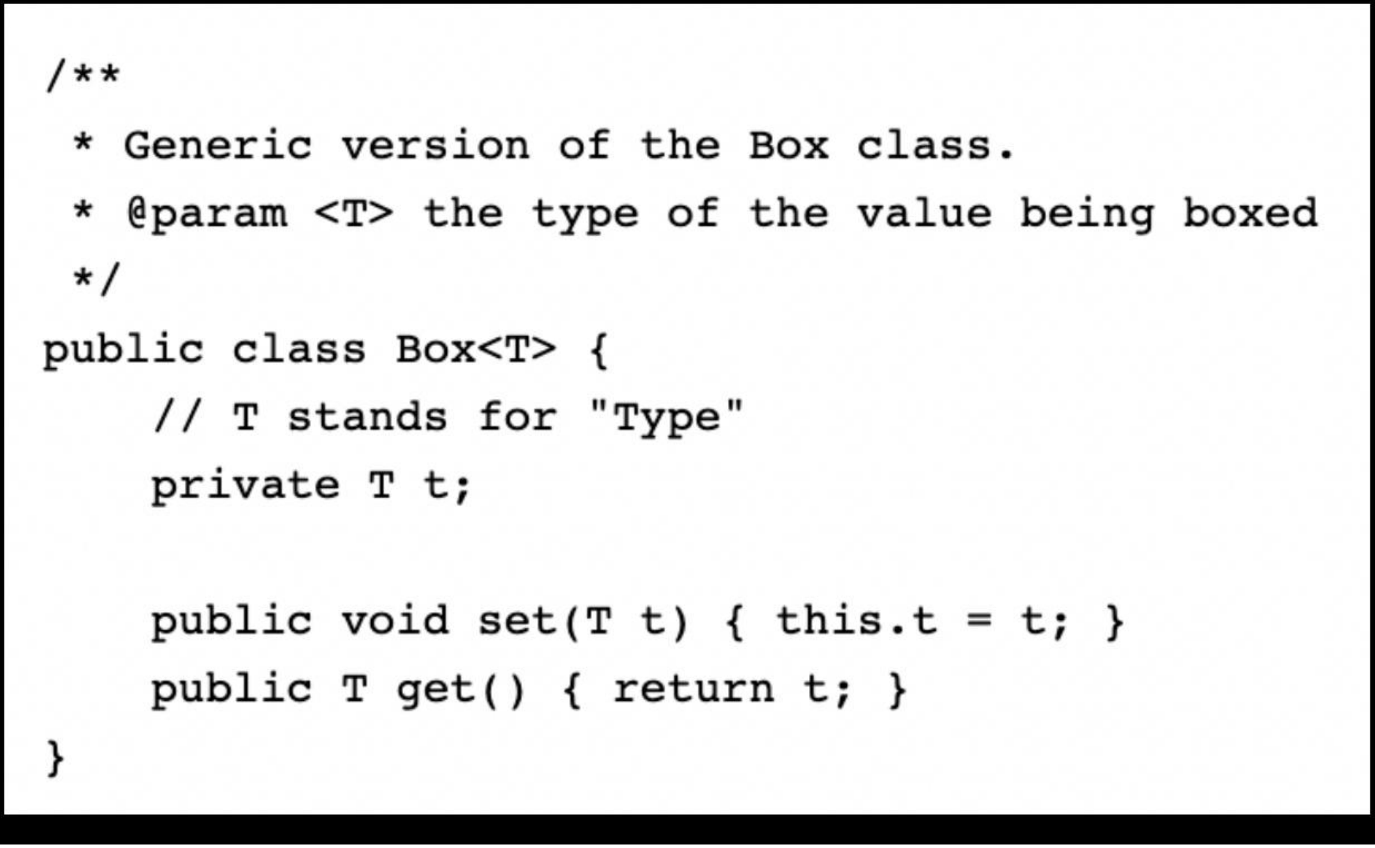
Example

Non Generic Box Class



```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Generic Box class



```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Instantiation of Generic Type Class

Unbounded type or raw type

```
1) Box obj = new Box();
```

→ Create Generic type as "Object"

bounded type

```
2) Box <Integer> obj = new Box <Integer> ();
```

redundant

↳ takes generic type as Integer

↳ If constructor can get object type during compile time

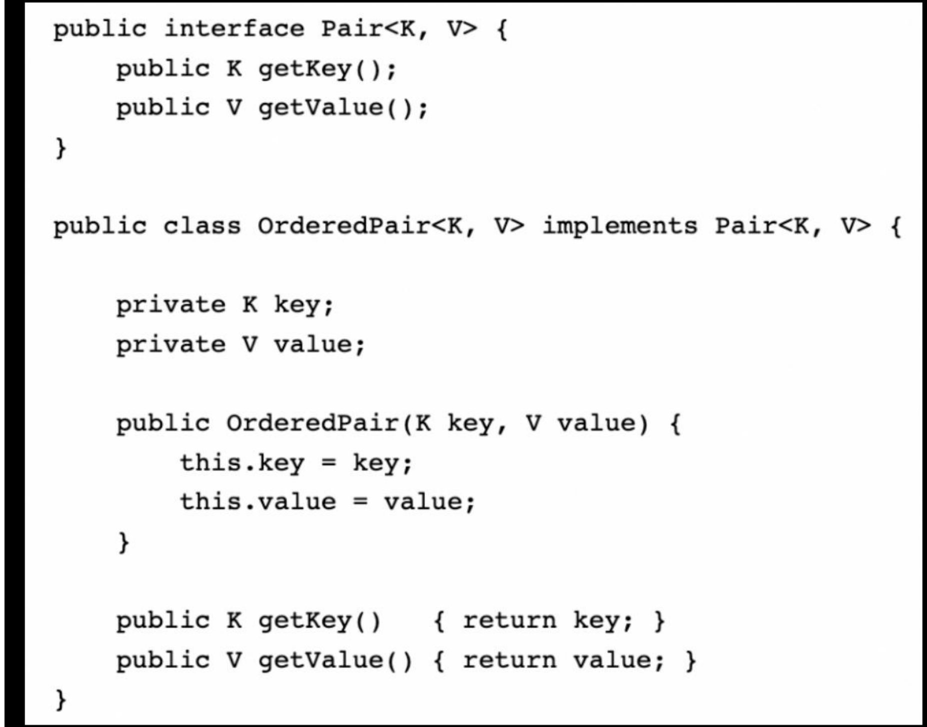
diamond syntax

```
3) Box <Integer> obj = new Box <> ();
```

↷ constructor invokation

↳ diamond

Multiple Type Parameters



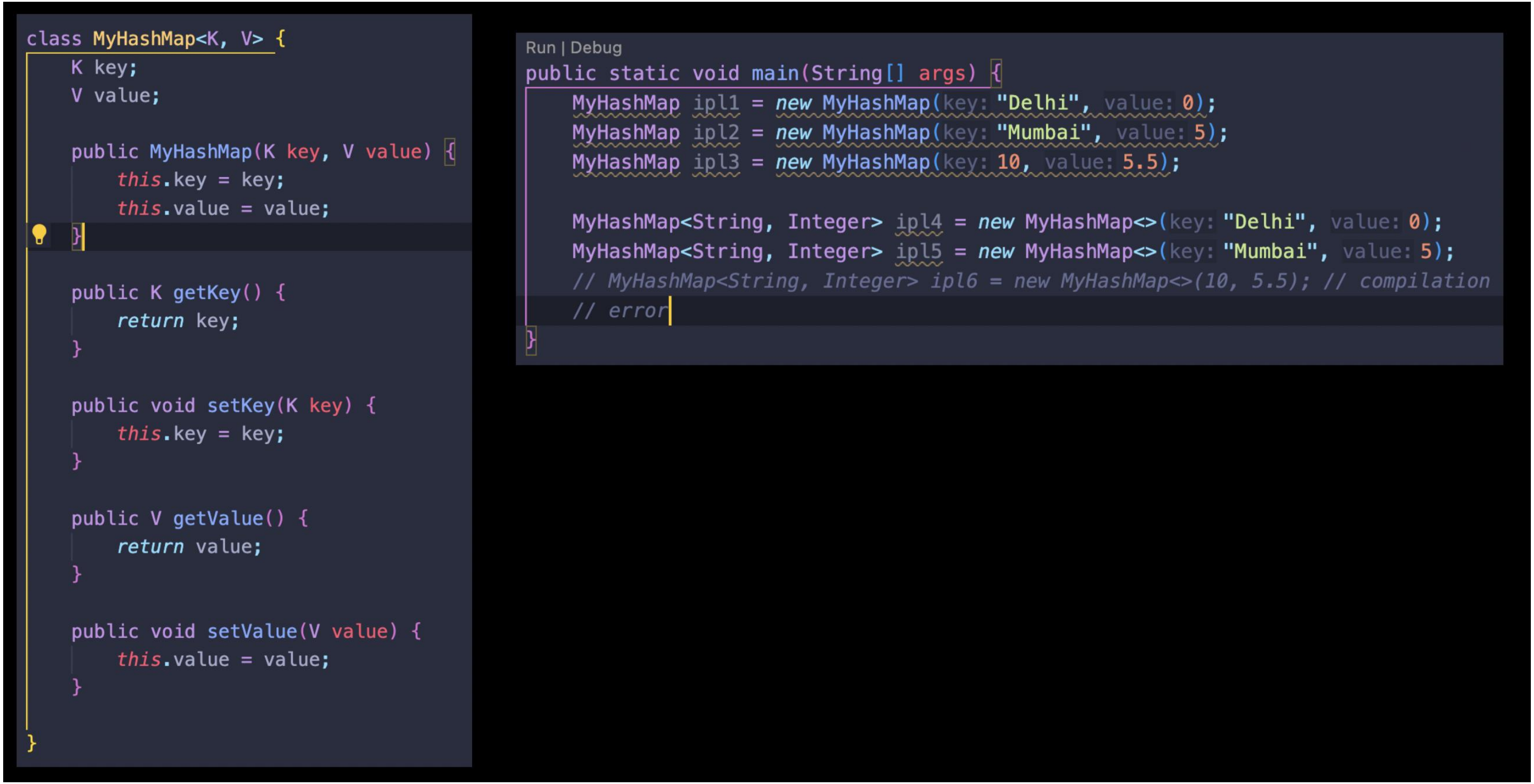
```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Object Creat^n

Pair<String, Integer> p
= new OrderedPair<>();

Parameterized type as type parameter

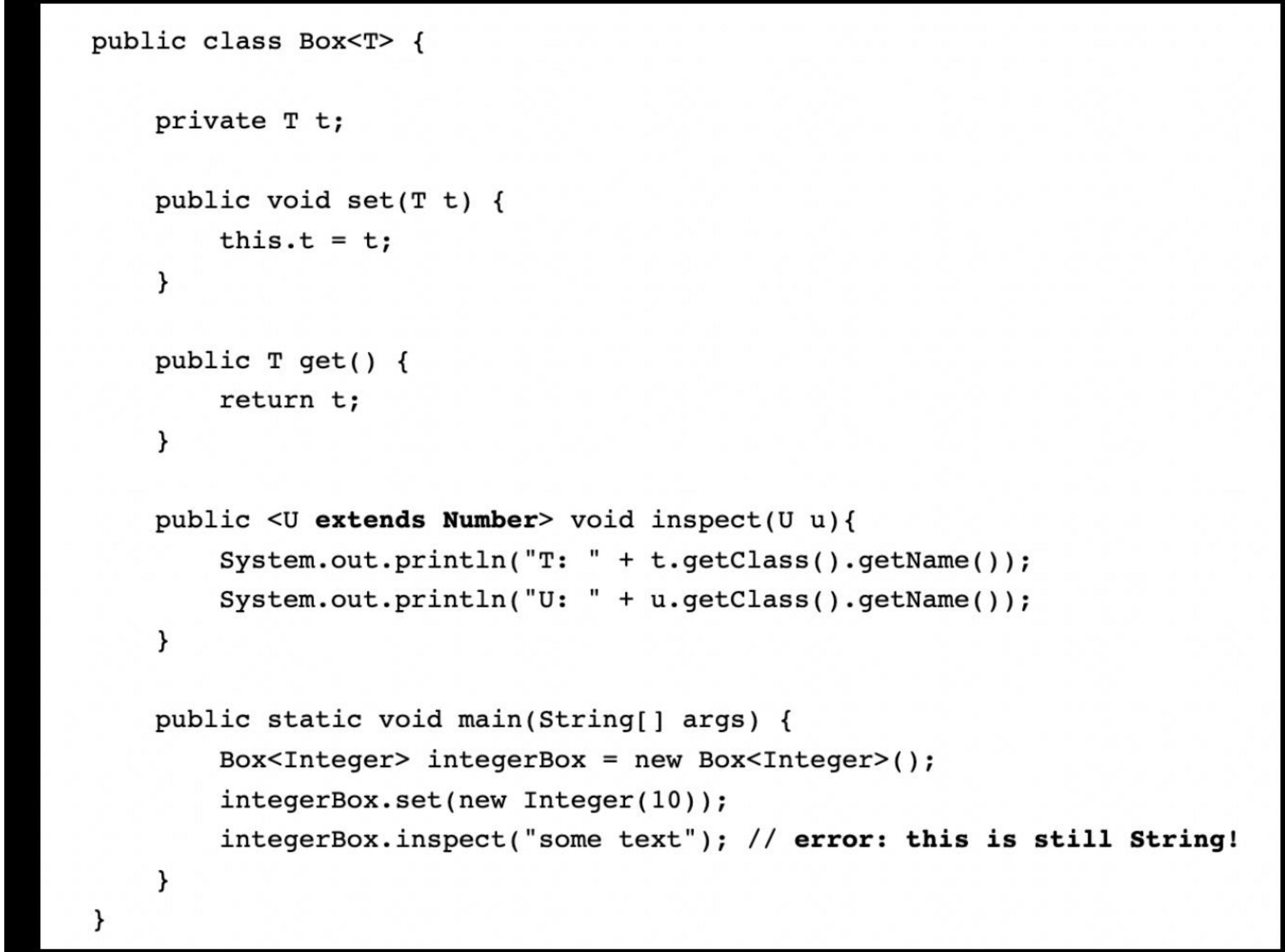
Pair<String, Arraylist<Integer>>
p = new OrderedPair<>();



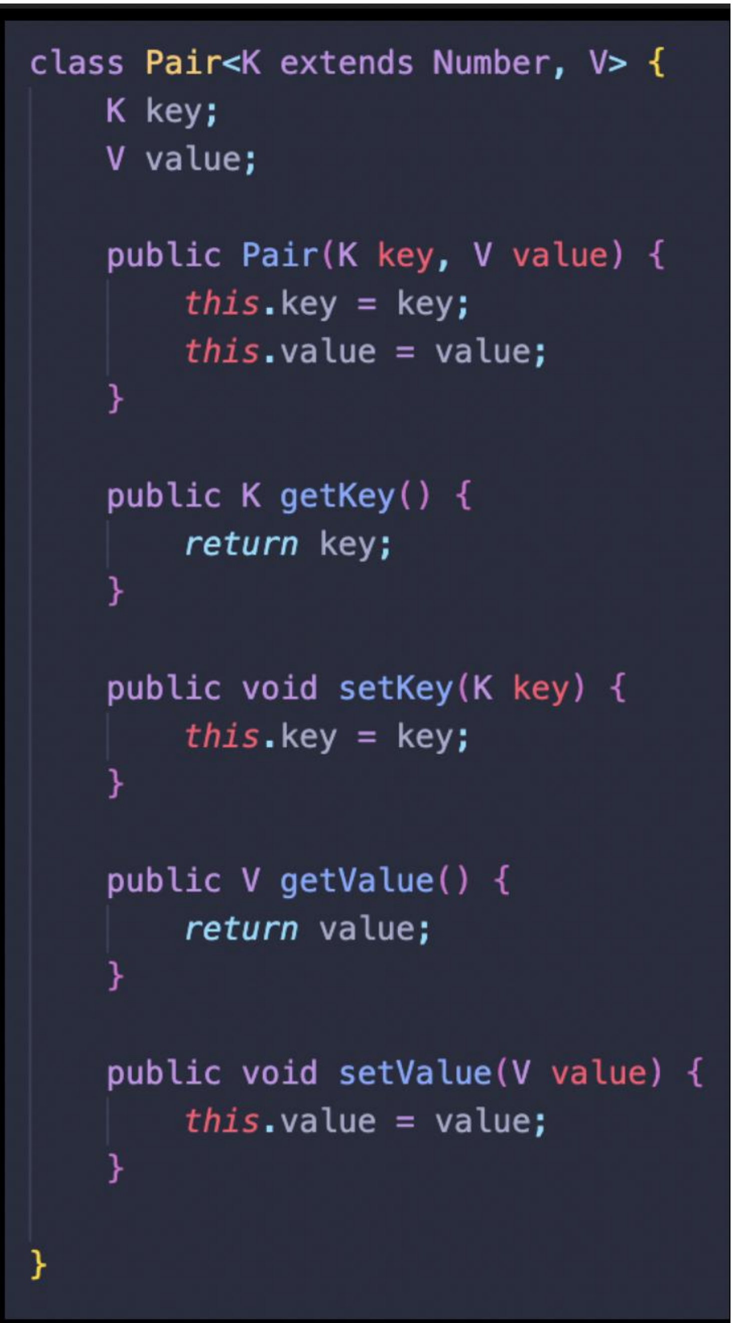
```java
class MyHashMap<K, V> {
    K key;
    V value;

    public MyHashMap(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }

}
```

```java
Run | Debug
public static void main(String[] args) {
    MyHashMap ipl1 = new MyHashMap(key: "Delhi", value: 0);
    MyHashMap ipl2 = new MyHashMap(key: "Mumbai", value: 5);
    MyHashMap ipl3 = new MyHashMap(key: 10, value: 5.5);

    MyHashMap<String, Integer> ipl4 = new MyHashMap<>(key: "Delhi", value: 0);
    MyHashMap<String, Integer> ipl5 = new MyHashMap<>(key: "Mumbai", value: 5);
    // MyHashMap<String, Integer> ipl6 = new MyHashMap<>(10, 5.5); // compilation
    // error
}
```

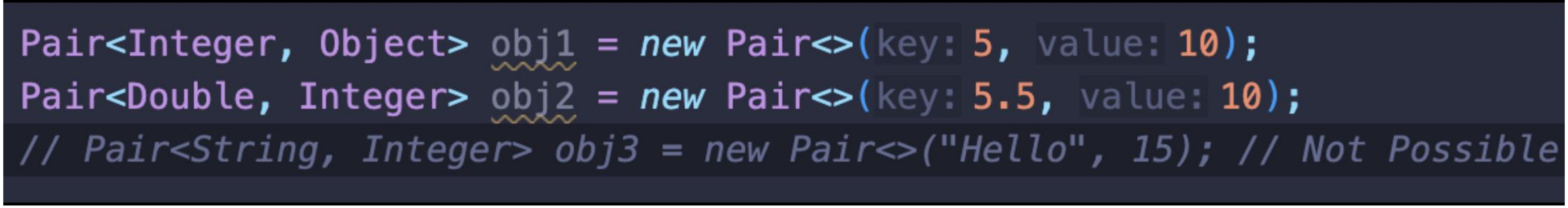
Bounded Type Parameters



```java
public class Box<T> {

    private T t;

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

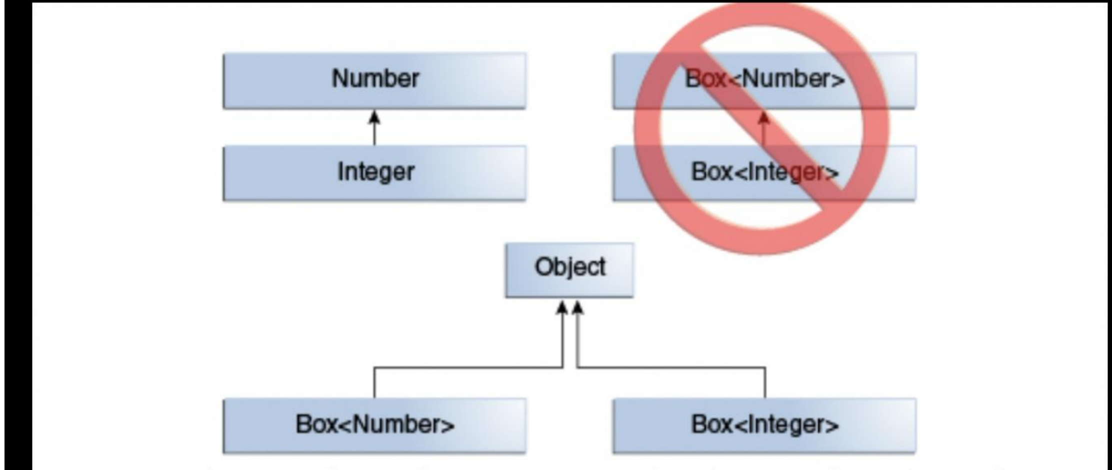


```java
class Pair<K extends Number, V> {
    K key;
    V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public void setKey(K key) {
        this.key = key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V value) {
        this.value = value;
    }

}
```



```java
Pair<Integer, Object> obj1 = new Pair<>(key: 5, value: 10);
Pair<Double, Integer> obj2 = new Pair<>(key: 5.5, value: 10);
// Pair<String, Integer> obj3 = new Pair<>("Hello", 15); // Not Possible
```

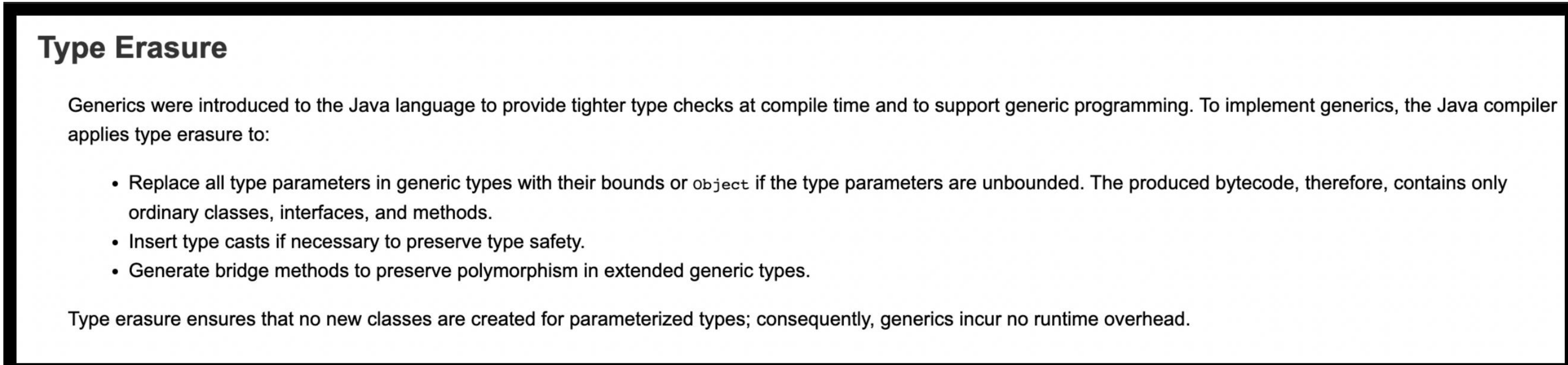
Generics & Inheritance



`Box<Integer>` is not a subtype of `Box<Number>` even though `Integer` is a subtype of `Number`.

eg

```
Box<Number> b = new Box<>();

b.add(new Integer(10)); ✓
b.add(new Double(5.5)); ✓

Box<Integer> b = new Box<>();
b.add(new Object()); ✗
b.add(new Integer(10)); ✓
b.add(new Double(5.5)); ✗
```

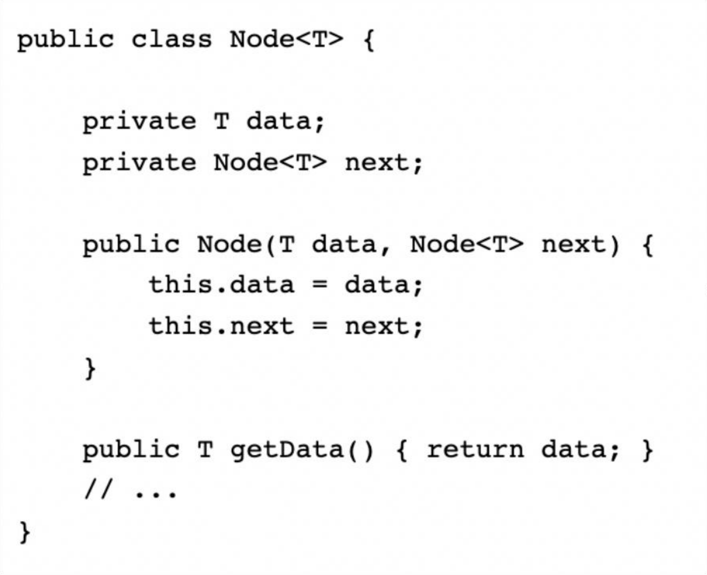


Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

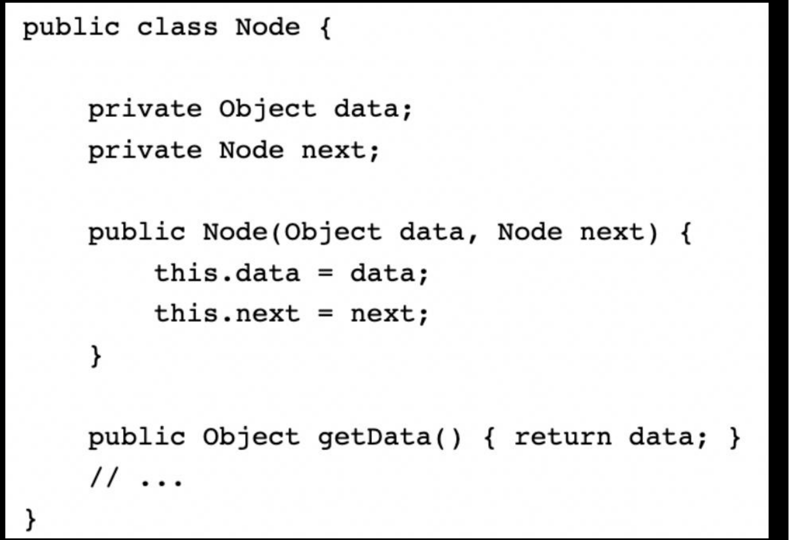


```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Unbounded type parameters

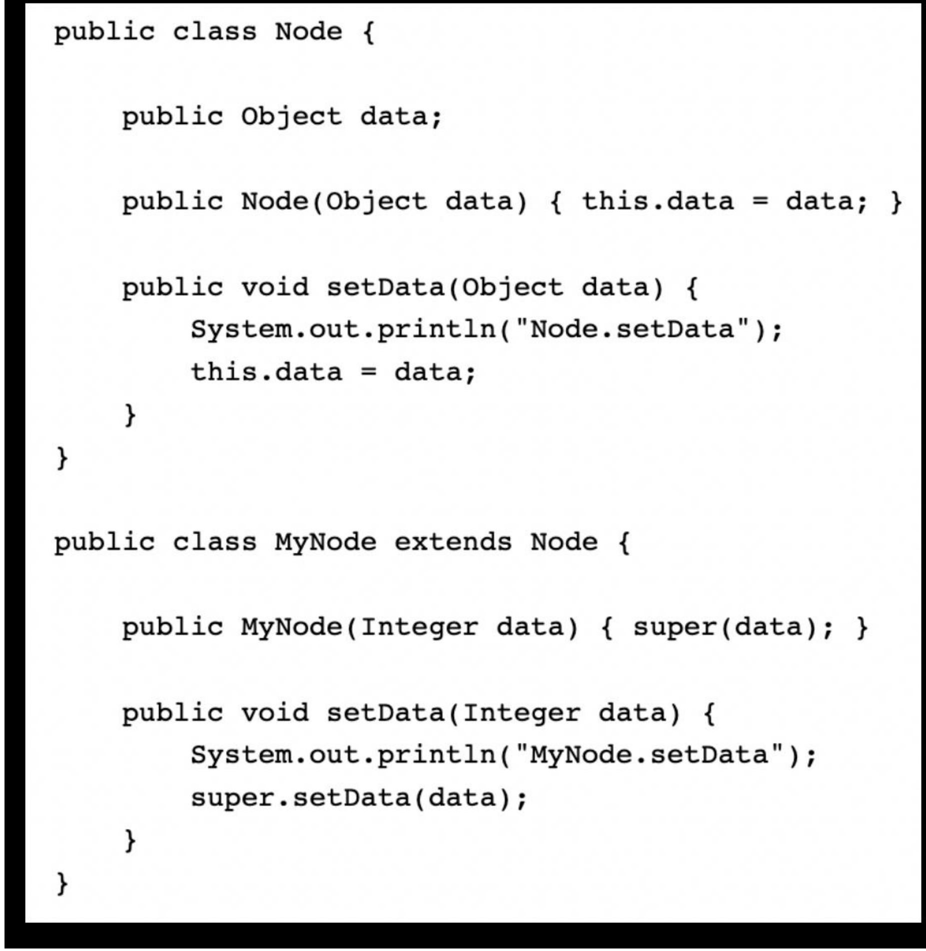
Compilatn { bytecode }

Node head = new Node();

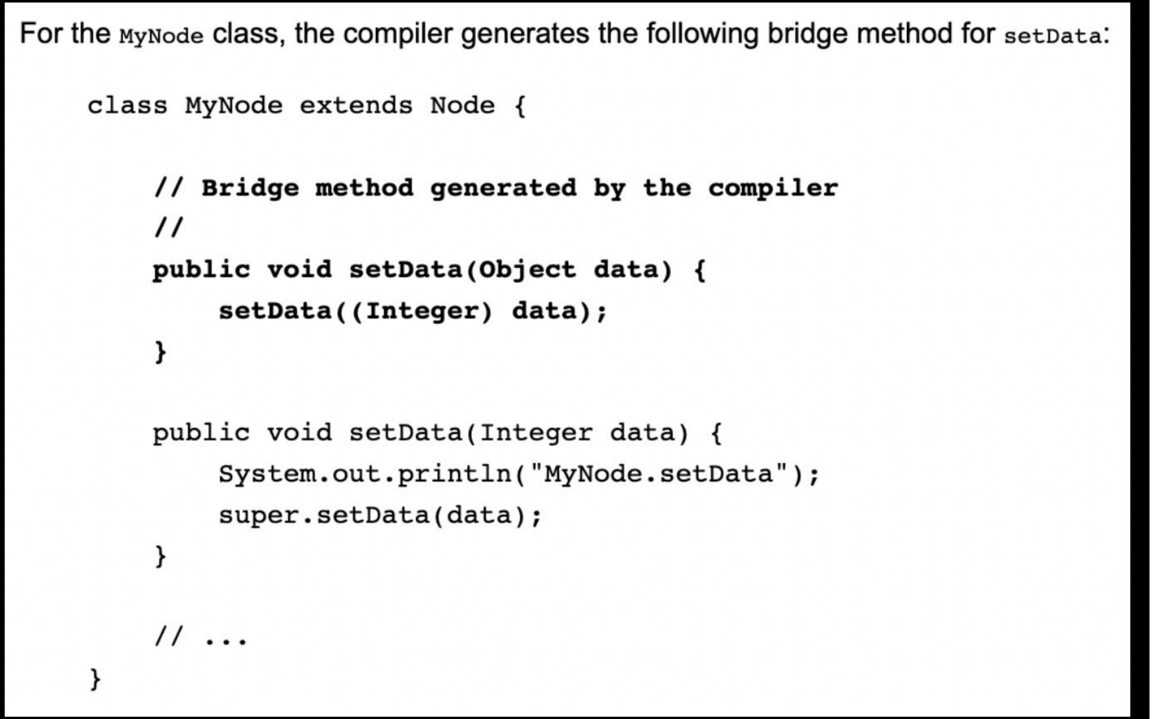


```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

Bridge Methods in Type Erasure



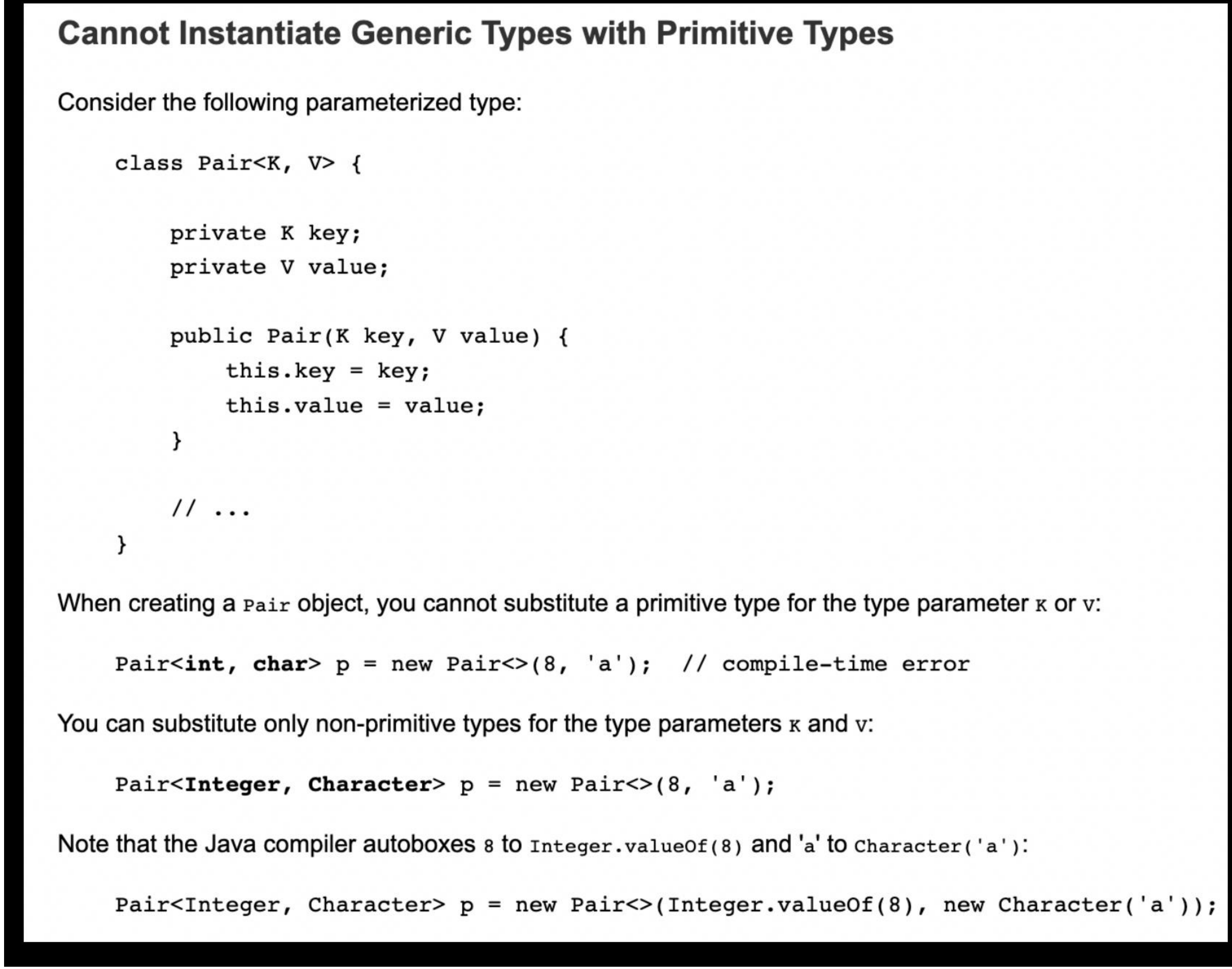
```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```



For the `MyNode` class, the compiler generates the following bridge method for `setData`:

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

Restrictions on Generics #1



Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

When creating a `Pair` object, you cannot substitute a primitive type for the type parameter `K` or `V`:

```
Pair<int, char> p = new Pair<>(8, 'a');  // compile-time error
```

You can substitute only non-primitive types for the type parameters `K` and `V`:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes `8` to `Integer.valueOf(8)` and 'a' to `Character('a')`:

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

Primitives as generics are not possible!

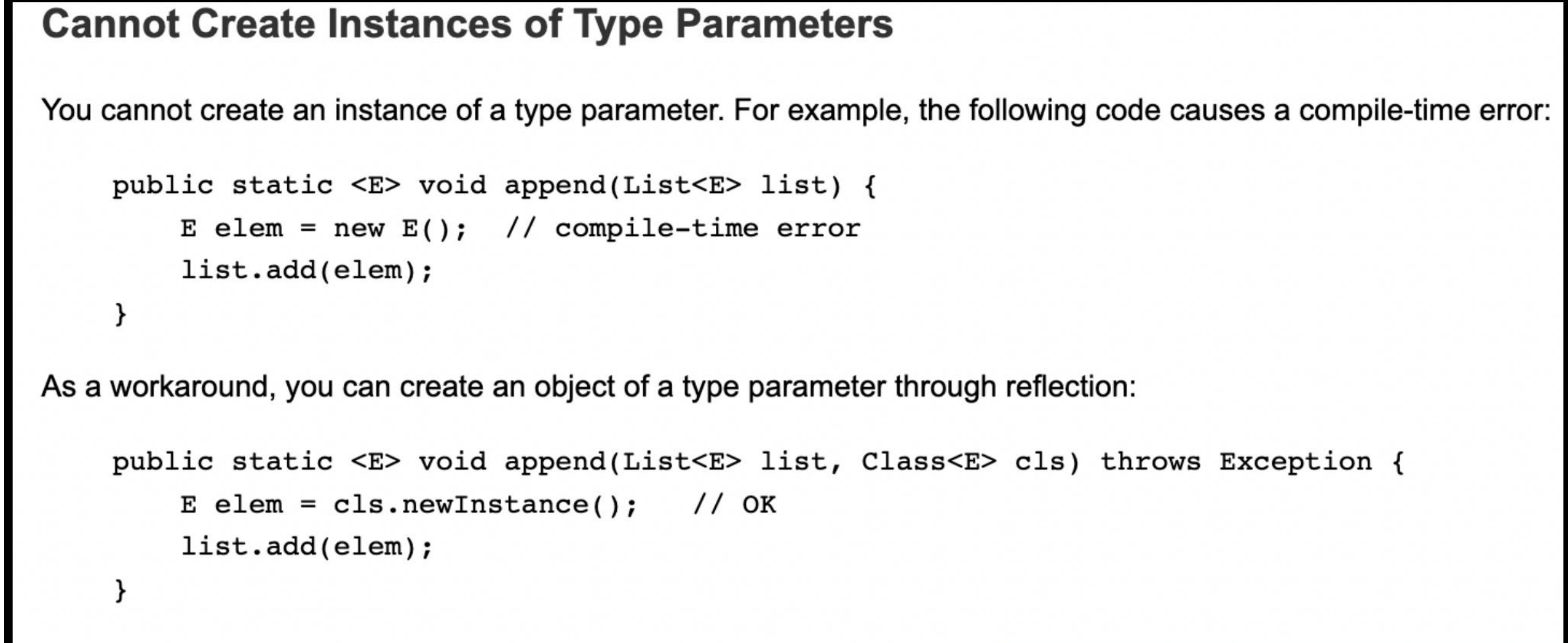
Arraylist ~~<int>~~ a

= new Arraylist<>();

a.add(10);

↳ Integer (wrapper class)

Restrictions on Generics #2



Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

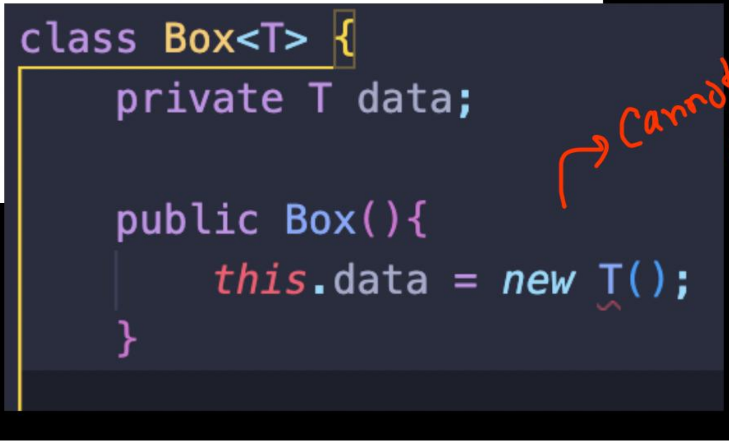
```
    public static <E> void append(List<E> list) {
        E elem = new E();  // compile-time error
        list.add(elem);
    }
```

As a workaround, you can create an object of a type parameter through reflection:

```
    public static <E> void append(List<E> list, Class<E> cls) throws Exception {
        E elem = cls.newInstance();   // OK
        list.add(elem);
    }
```

You can invoke the `append` method as follows:

```
    List<String> ls = new ArrayList<>();
    append(ls, String.class);
```



```
class Box<T> {
    private T data;

    public Box(){
        this.data = new T();
    }
```

Cannot create object of generic type
⇓
Compiletime error

Custom Arraylist

0	1	2	3	4	5	6	7	8	9
10	20	30	40	50	60	70	80	90	100

↓ cap + cap/2 = 10+10/2 = 15

10 20 30 40 50 60 70 80 90 100 110	120	130	140	150

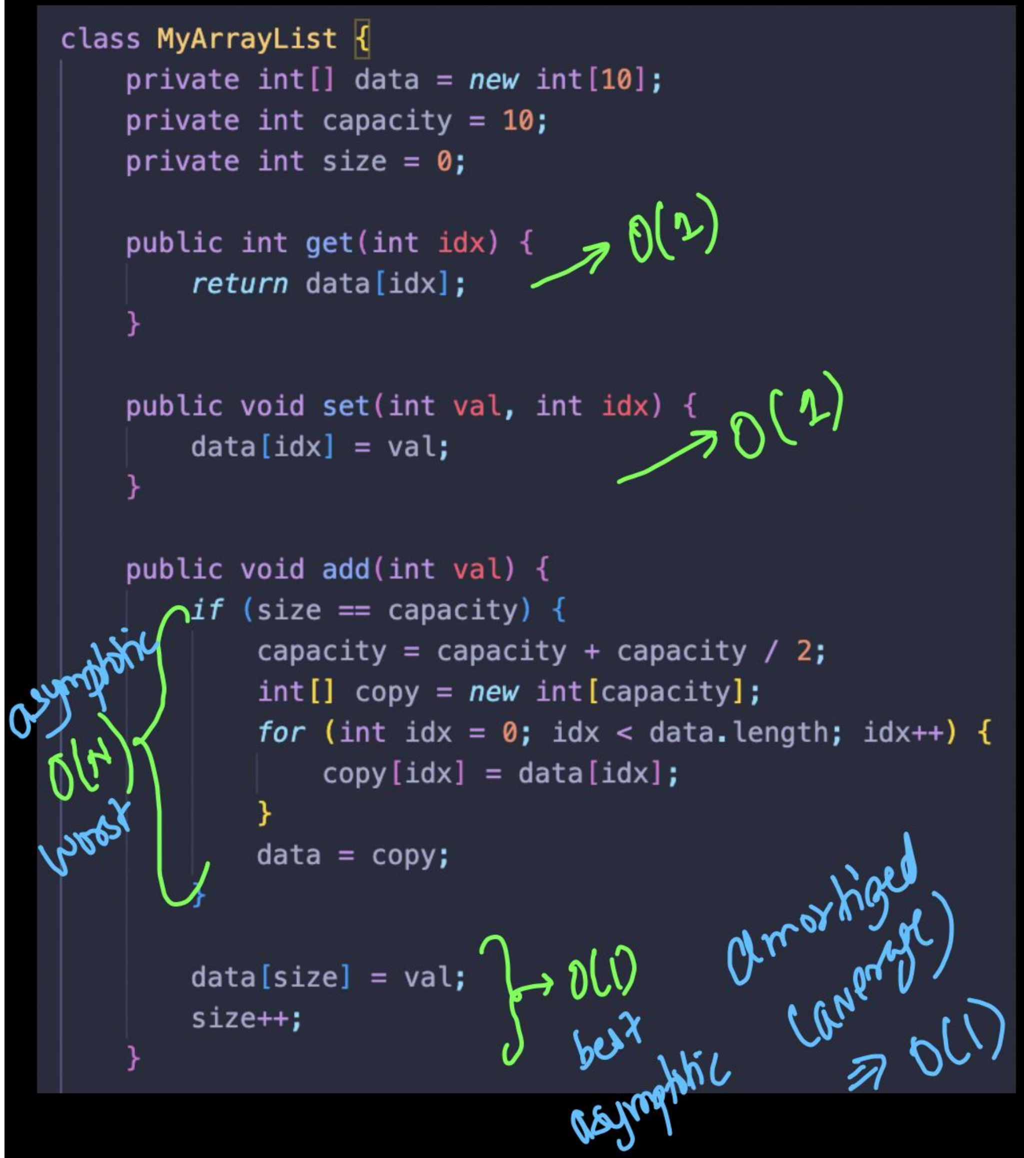
↓ Cap + cap/2 = 15+15/2 = 23

10 - 150 160 0 0 0 0 0 0

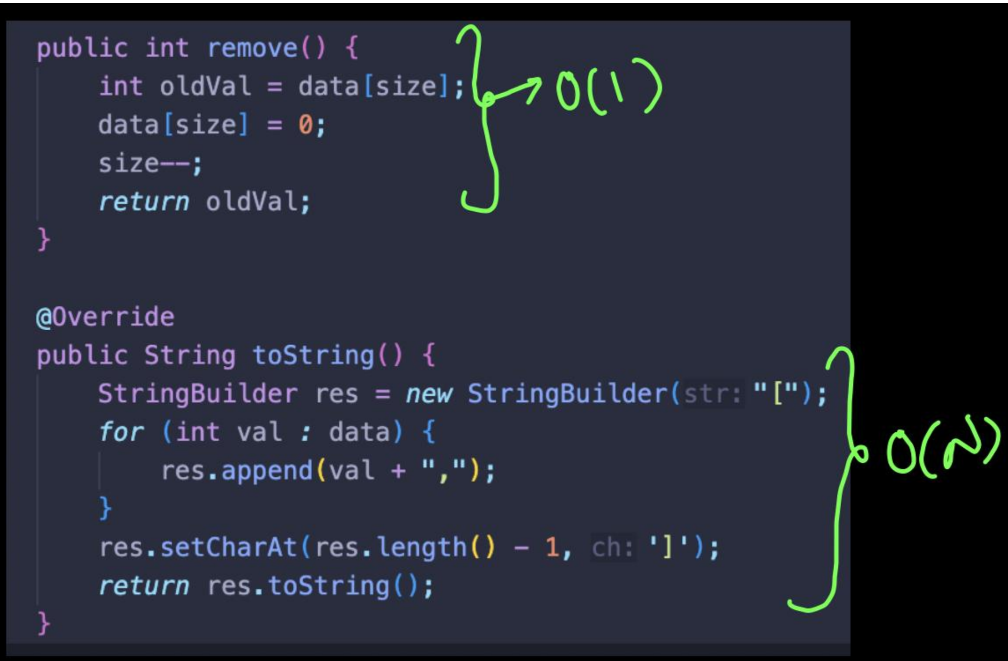
CAPACITY = ~~10~~ ~~15~~ 23

size = ~~0~~ ~~1~~ ~~2~~ ~~3~~ -- ~~9~~ ~~10~~ ~~11~~ ~~12~~ ~~13~~ ~~14~~ ~~15~~ 16

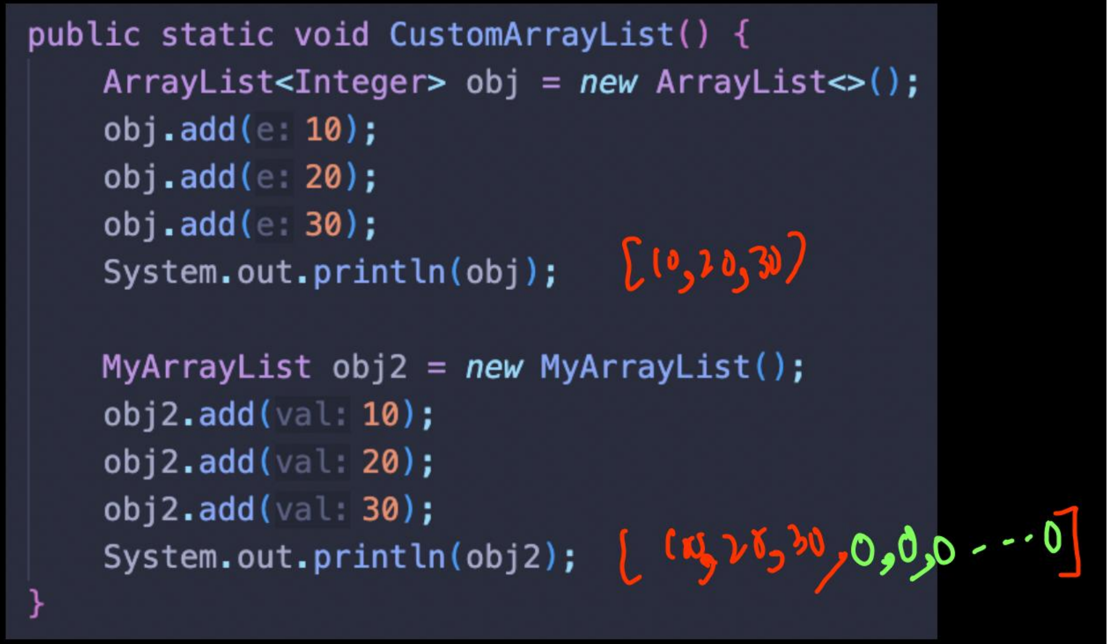
```
Arraylist<Integer>
   arr = new Arraylist<>();

for (int i= 10; i<=1000; i = i+10){
    arr.add(i);

}

for(int i=
```



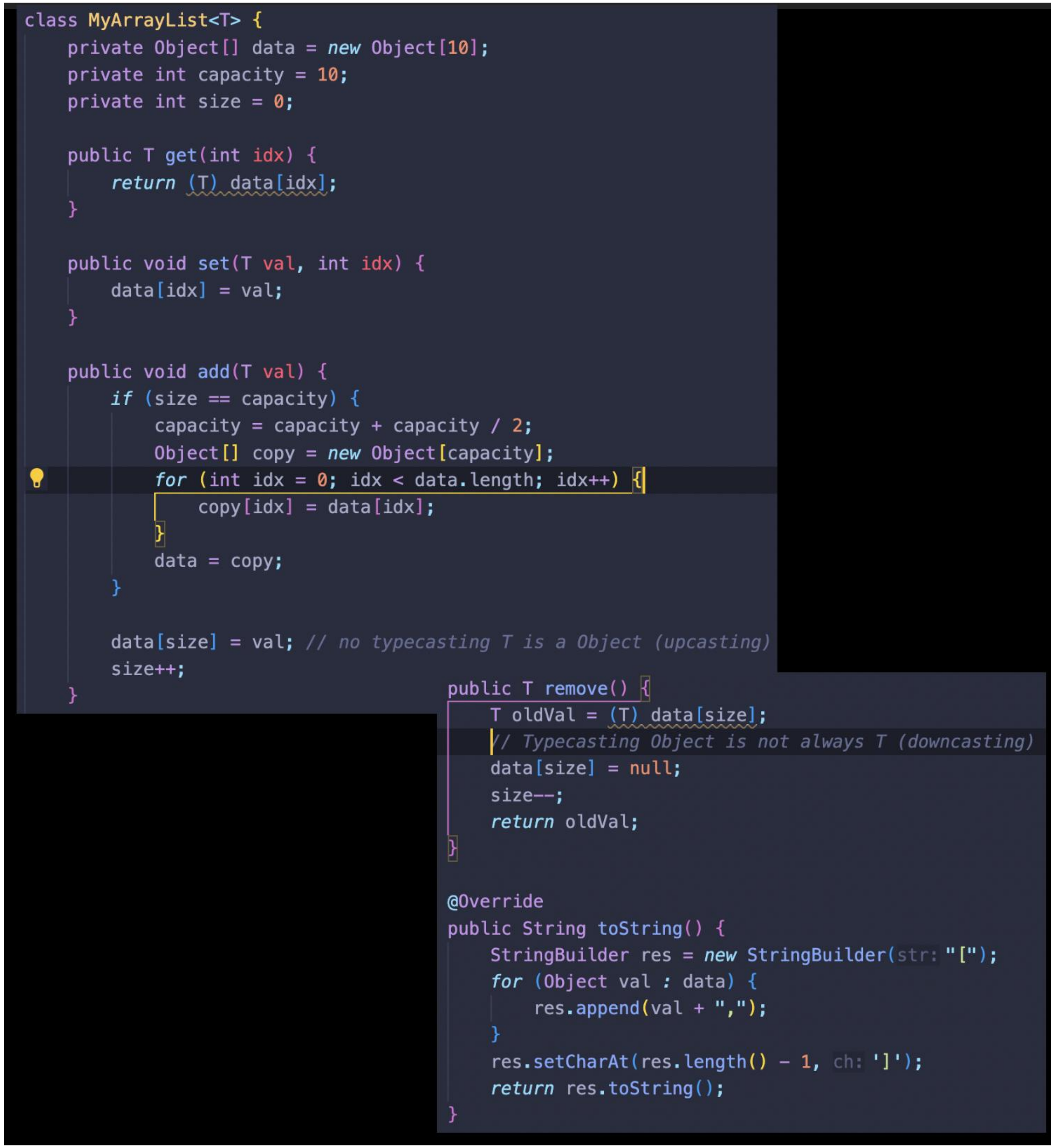
```java
class MyArrayList {
    private int[] data = new int[10];
    private int capacity = 10;
    private int size = 0;

    public int get(int idx) {
        return data[idx];
    }

    public void set(int val, int idx) {
        data[idx] = val;
    }

    public void add(int val) {
        if (size == capacity) {
            capacity = capacity + capacity / 2;
            int[] copy = new int[capacity];
            for (int idx = 0; idx < data.length; idx++) {
                copy[idx] = data[idx];
            }
            data = copy;
        }

        data[size] = val;
        size++;
    }
```



```java
public int remove() {
    int oldVal = data[size];
    data[size] = 0;
    size--;
    return oldVal;
}

@Override
public String toString() {
    StringBuilder res = new StringBuilder(str: "[");
    for (int val : data) {
        res.append(val + ",");
    }
    res.setCharAt(res.length() - 1, ch: ']');
    return res.toString();
}
```



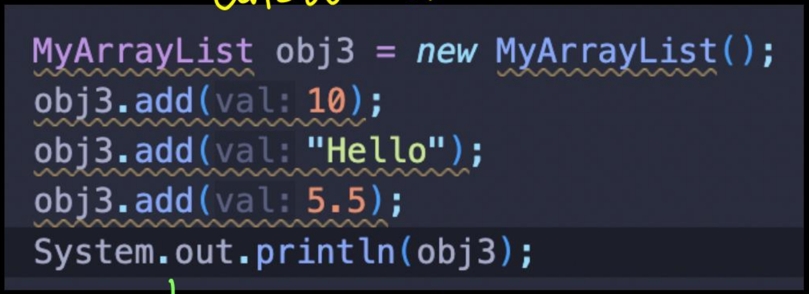
```java
public static void CustomArrayList() {
    ArrayList<Integer> obj = new ArrayList<>();
    obj.add(e: 10);
    obj.add(e: 20);
    obj.add(e: 30);
    System.out.println(obj);

    MyArrayList obj2 = new MyArrayList();
    obj2.add(val: 10);
    obj2.add(val: 20);
    obj2.add(val: 30);
    System.out.println(obj2);
}
```



```java
class MyArrayList<T> {
    private Object[] data = new Object[10];
    private int capacity = 10;
    private int size = 0;

    public T get(int idx) {
        return (T) data[idx];
    }

    public void set(T val, int idx) {
        data[idx] = val;
    }

    public void add(T val) {
        if (size == capacity) {
            capacity = capacity + capacity / 2;
            Object[] copy = new Object[capacity];
            for (int idx = 0; idx < data.length; idx++) {
                copy[idx] = data[idx];
            }
            data = copy;
        }

        data[size] = val; // no typecasting T is a Object (upcasting)
        size++;
    }
```

```java
    public T remove() {
        T oldVal = (T) data[size];
        // Typecasting Object is not always T (downcasting)
        data[size] = null;
        size--;
        return oldVal;
    }

    @Override
    public String toString() {
        StringBuilder res = new StringBuilder(str: "[");
        for (Object val : data) {
            res.append(val + ",");
        }
        res.setCharAt(res.length() - 1, ch: ']');
        return res.toString();
    }
```

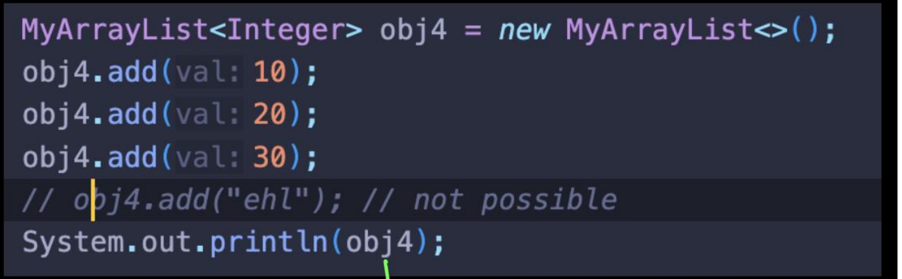
unbounded



```java
MyArrayList obj3 = new MyArrayList();
obj3.add(val: 10);
obj3.add(val: "Hello");
obj3.add(val: 5.5);
System.out.println(obj3);
```

[10, "Hello", 5.5, N, N, N, - - N]

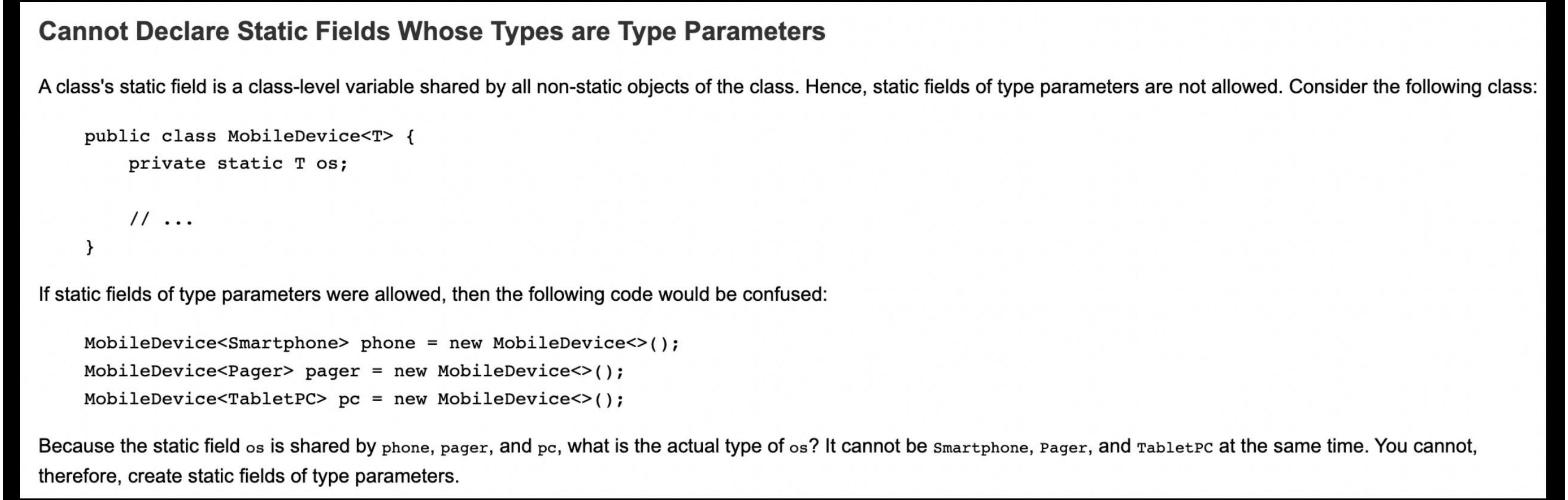
bounded



```java
MyArrayList<Integer> obj4 = new MyArrayList<>();
obj4.add(val: 10);
obj4.add(val: 20);
obj4.add(val: 30);
// obj4.add("ehl"); // not possible
System.out.println(obj4);
```

[10, 20, 30, N, N, N - - - N]

Restrictions on Generics #3



Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

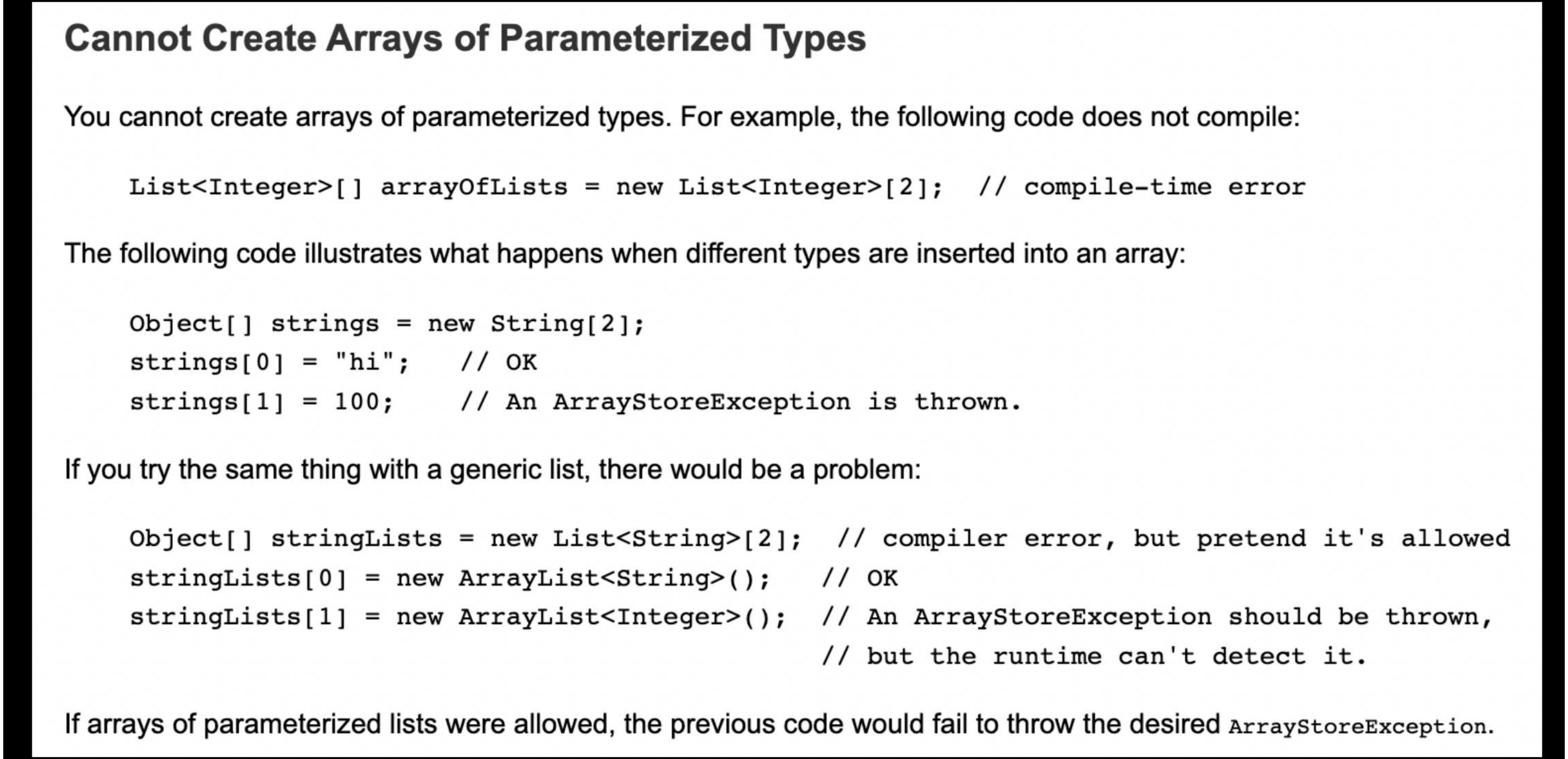
```
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field `os` is shared by `phone`, `pager`, and `pc`, what is the actual type of `os`? It cannot be `Smartphone`, `Pager`, and `TabletPC` at the same time. You cannot, therefore, create static fields of type parameters.

Restrictions on Generics #4



Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

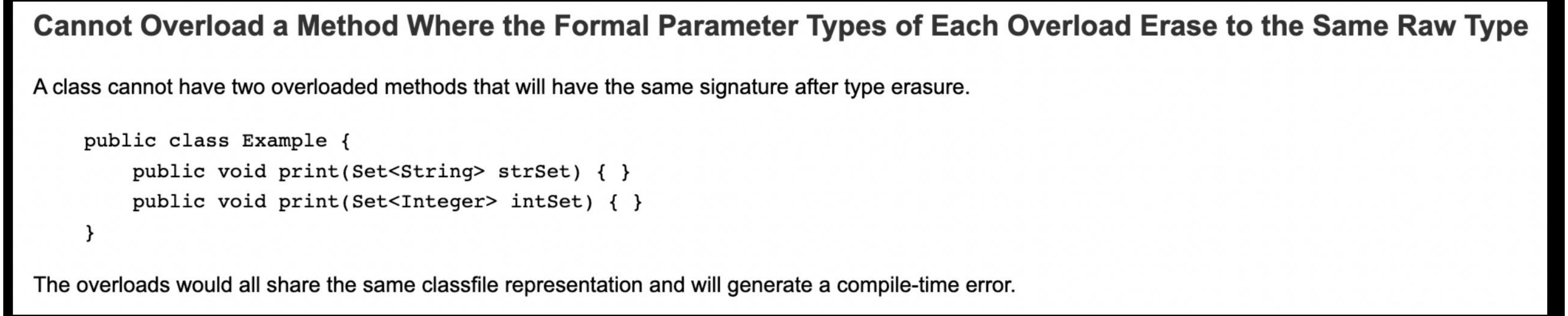
```
Object[] strings = new String[2];
strings[0] = "hi";    // OK
strings[1] = 100;     // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[2];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired `ArrayStoreException`.

Restrictions on Generics #5



Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.