Object-Oriented Programming in Java

Generics

Generics - Background

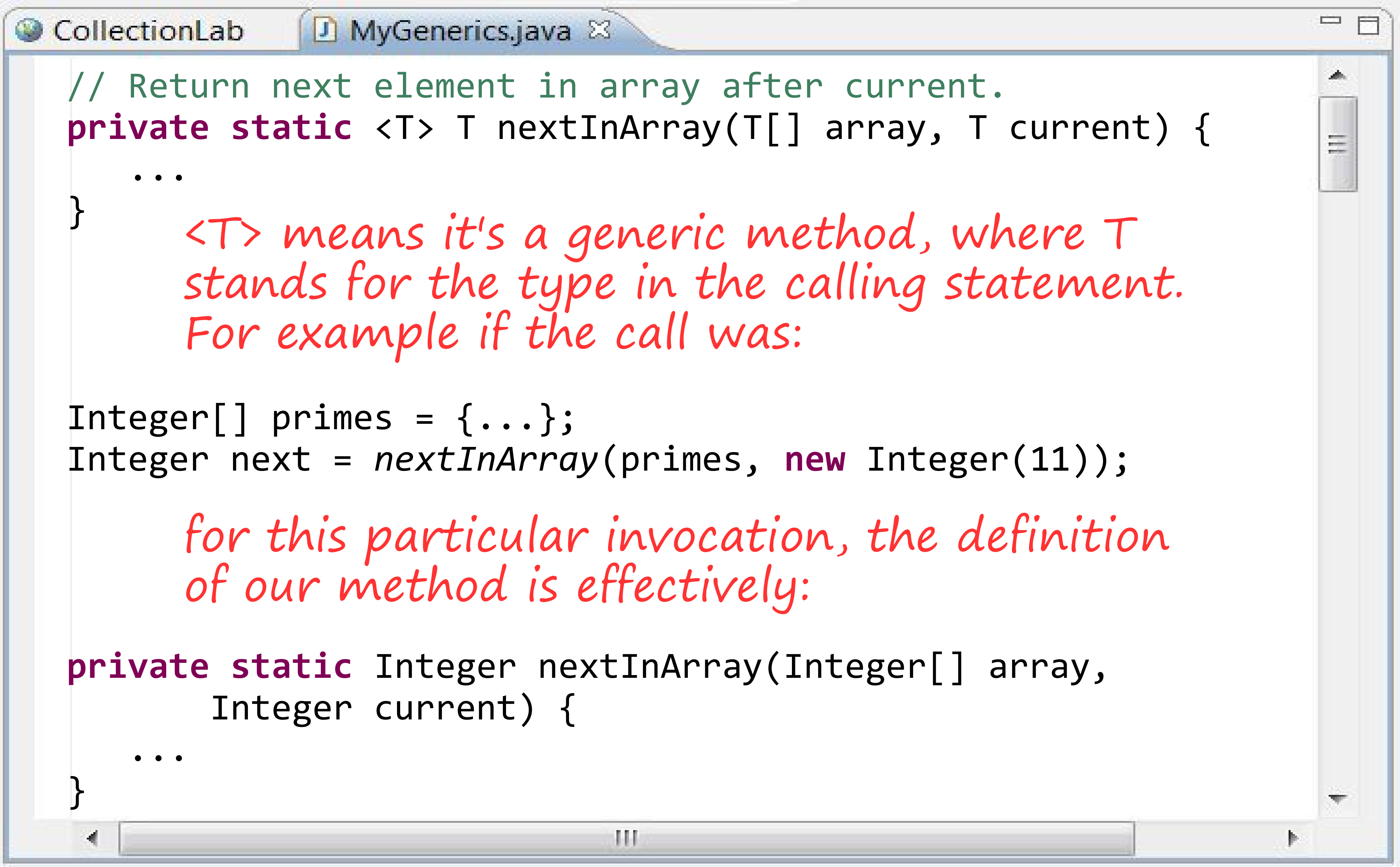
Methods have parameters, which allow them to be called with different values. E.g.

```
public double sqrt(double d) {
   ...
}
```

returns the square root of whatever value is passed as a parameter at run-time.

With **Generics**, a method can be called with different **types**, as well as different values.

A Generic Method



```
// Return next element in array after current.
private static <T> T nextInArray(T[] array, T current) {
    ...
}
```

<T> means it's a generic method, where T stands for the type in the calling statement. For example if the call was:

```
Integer[] primes = {...};
Integer next = nextInArray(primes, new Integer(11));
```

for this particular invocation, the definition of our method is effectively:

```
private static Integer nextInArray(Integer[] array,
        Integer current) {
    ...
}
```

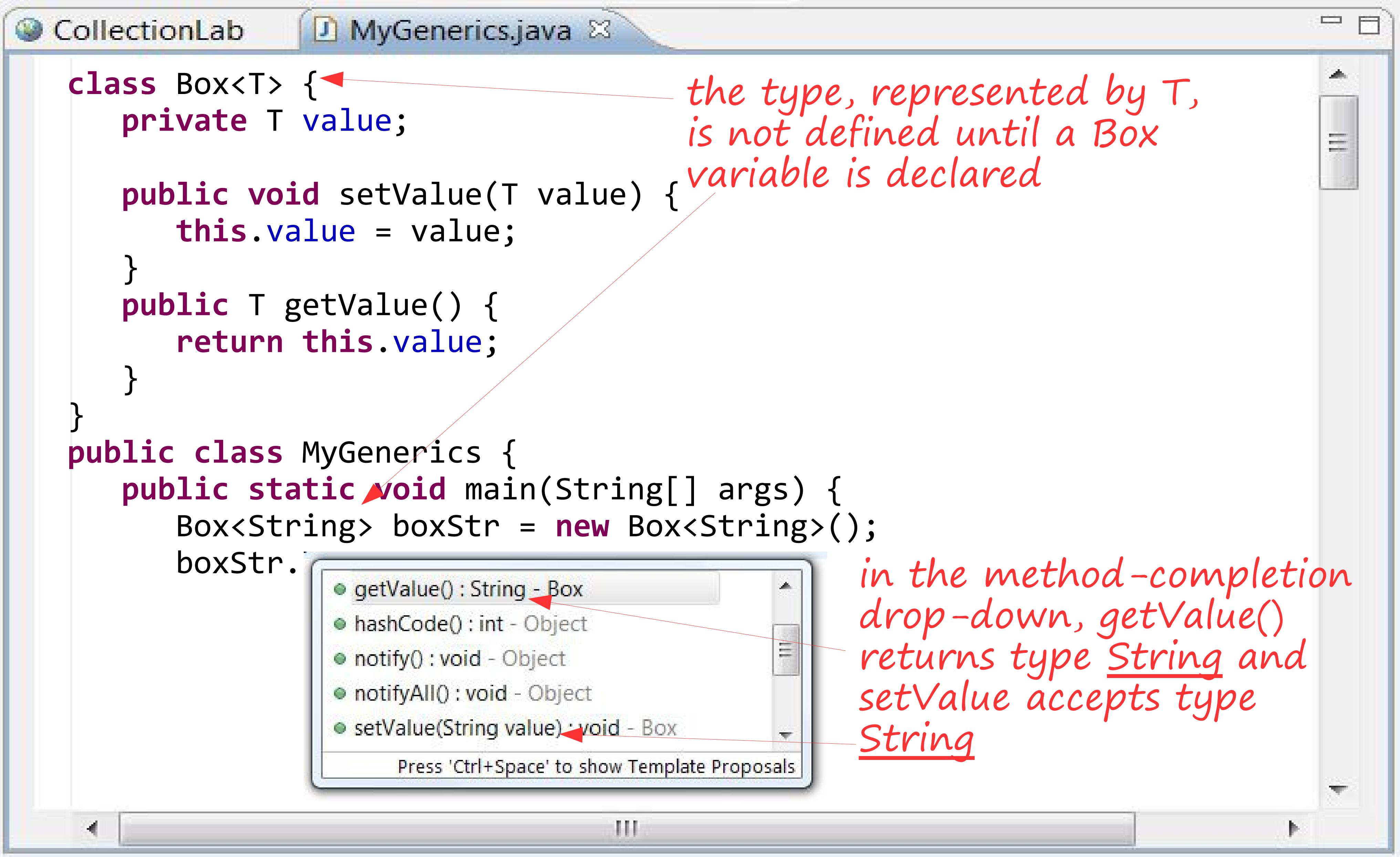
Generic Class

Generic types can be applied to a whole class:

```
class Box<T> {
    ...
}
```

- The **<>** shows it's a Generic class
- **T** stands for the variable type
- By convention the name of the type is upper case

Box Class with type String



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

   public void setValue(T value) {
      this.value = value;
   }
   public T getValue() {
      return this.value;
   }
}
public class MyGenerics {
   public static void main(String[] args) {
      Box<String> boxStr = new Box<String>();
      boxStr.
```

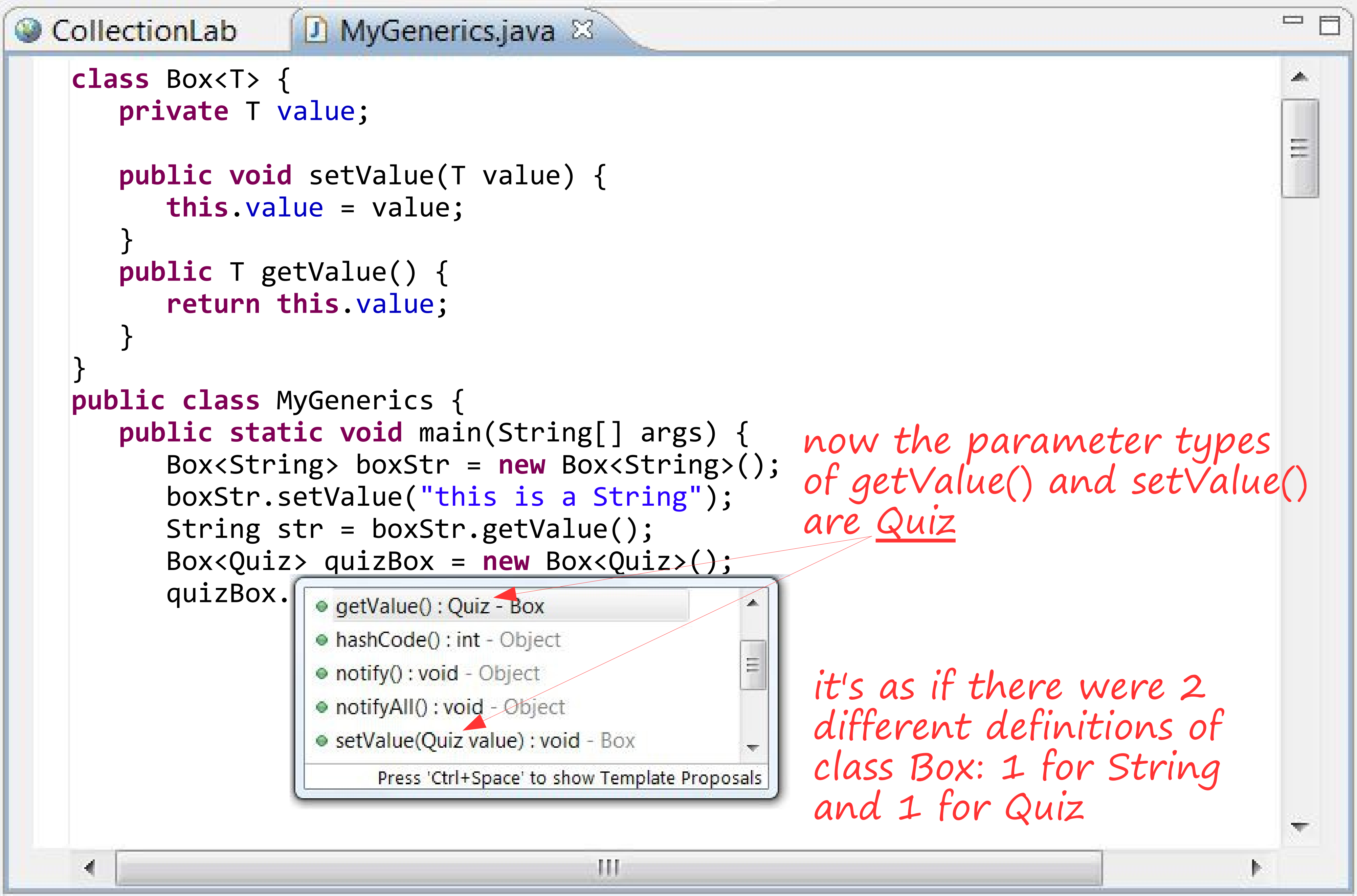
- getValue() : String - Box
- hashCode() : int - Object
- notify() : void - Object
- notifyAll() : void - Object
- setValue(String value) : void - Box

Press 'Ctrl+Space' to show Template Proposals

the type, represented by T, is not defined until a Box variable is declared

in the method-completion drop-down, getValue() returns type String and setValue accepts type String

Box Class with type Quiz



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

   public void setValue(T value) {
      this.value = value;
   }
   public T getValue() {
      return this.value;
   }
}
public class MyGenerics {
   public static void main(String[] args) {
      Box<String> boxStr = new Box<String>();
      boxStr.setValue("this is a String");
      String str = boxStr.getValue();
      Box<Quiz> quizBox = new Box<Quiz>();
      quizBox.
```

```
getValue() : Quiz - Box
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
setValue(Quiz value) : void - Box
Press 'Ctrl+Space' to show Template Proposals
```

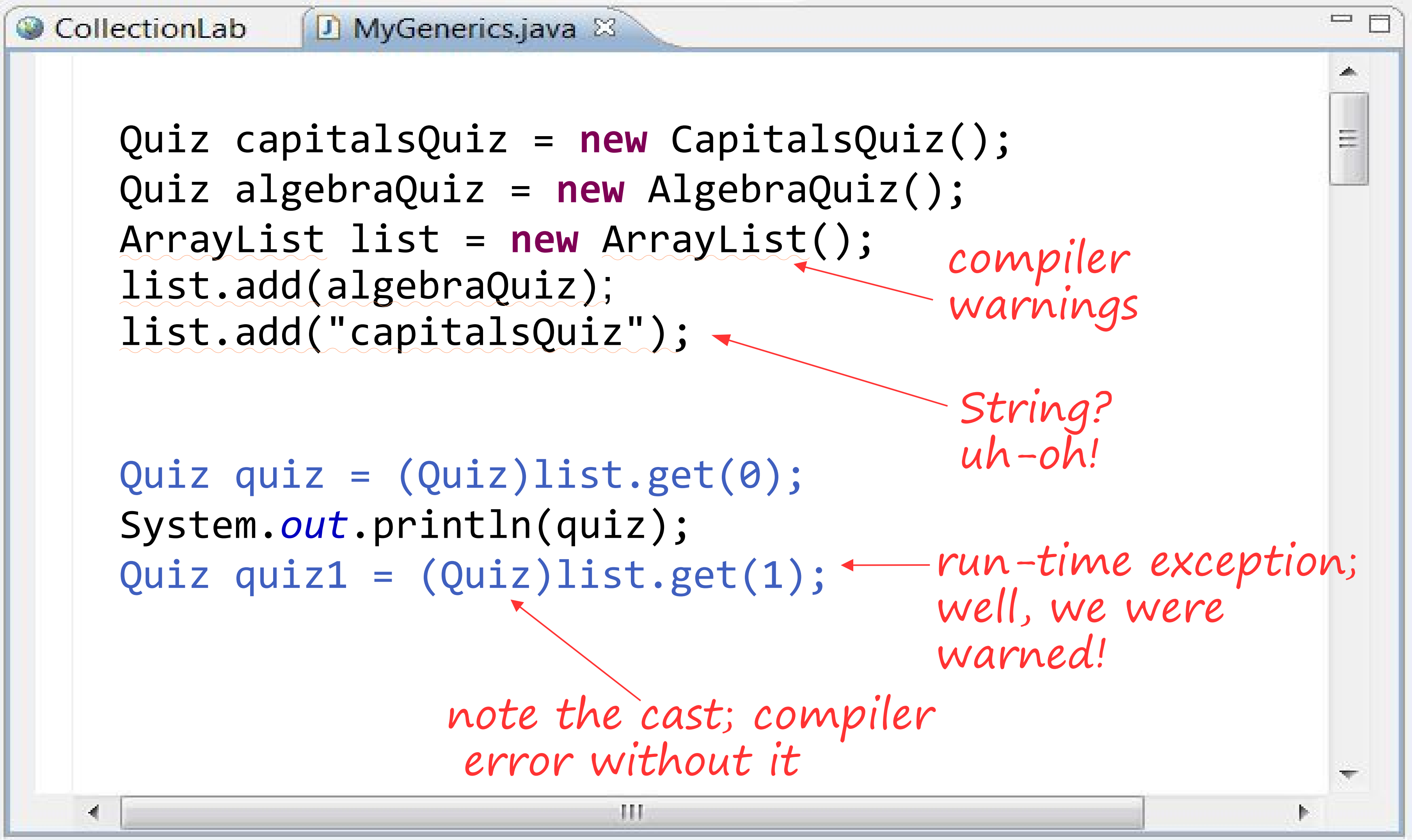
now the parameter types of getValue() and setValue() are Quiz

it's as if there were 2 different definitions of class Box: 1 for String and 1 for Quiz

Generics - Purpose

- provide compile-time type safety
- reduce risk of programmer errors
- are most often used with Collections

Using Collections without Generics



```java
Quiz capitalsQuiz = new CapitalsQuiz();
Quiz algebraQuiz = new AlgebraQuiz();
ArrayList list = new ArrayList();
list.add(algebraQuiz);
list.add("capitalsQuiz");


Quiz quiz = (Quiz)list.get(0);
System.out.println(quiz);
Quiz quiz1 = (Quiz)list.get(1);
```

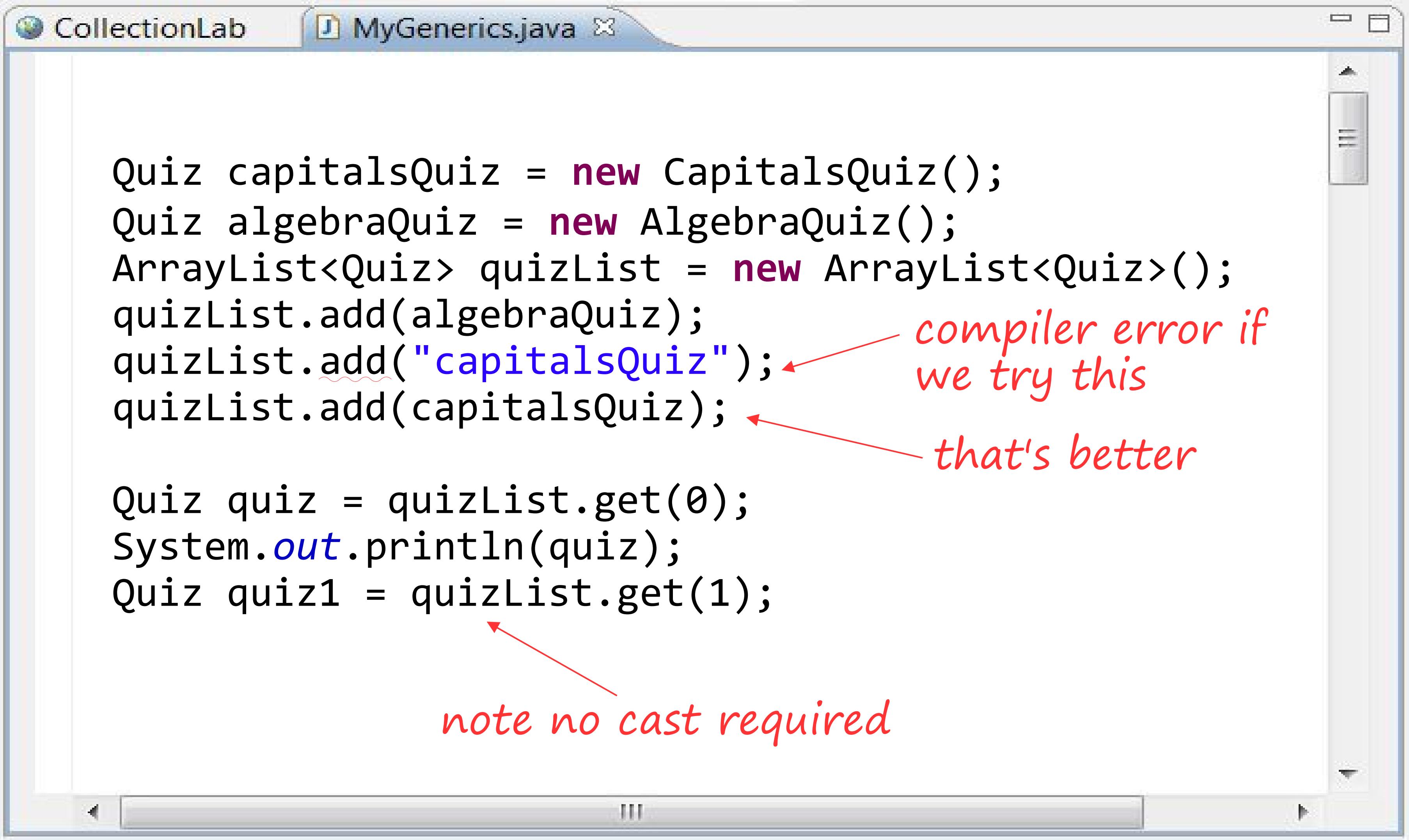
compiler warnings

String? uh-oh!

run-time exception; well, we were warned!

note the cast; compiler error without it

Using Collections with Generics
(the most common use of Generics)



```
Quiz capitalsQuiz = new CapitalsQuiz();
Quiz algebraQuiz = new AlgebraQuiz();
ArrayList<Quiz> quizList = new ArrayList<Quiz>();
quizList.add(algebraQuiz);
quizList.add("capitalsQuiz");
quizList.add(capitalsQuiz);

Quiz quiz = quizList.get(0);
System.out.println(quiz);
Quiz quiz1 = quizList.get(1);
```

compiler error if we try this

that's better

note no cast required

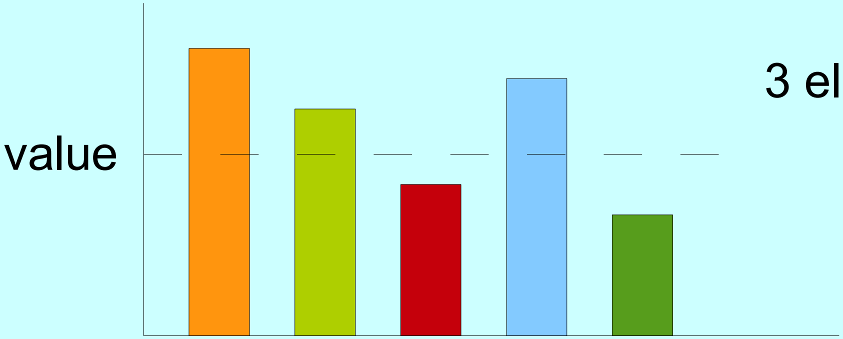
Other Uses of Generics

- Generic Methods
- Wild Cards
- User-defined Container Classes

Generic Methods

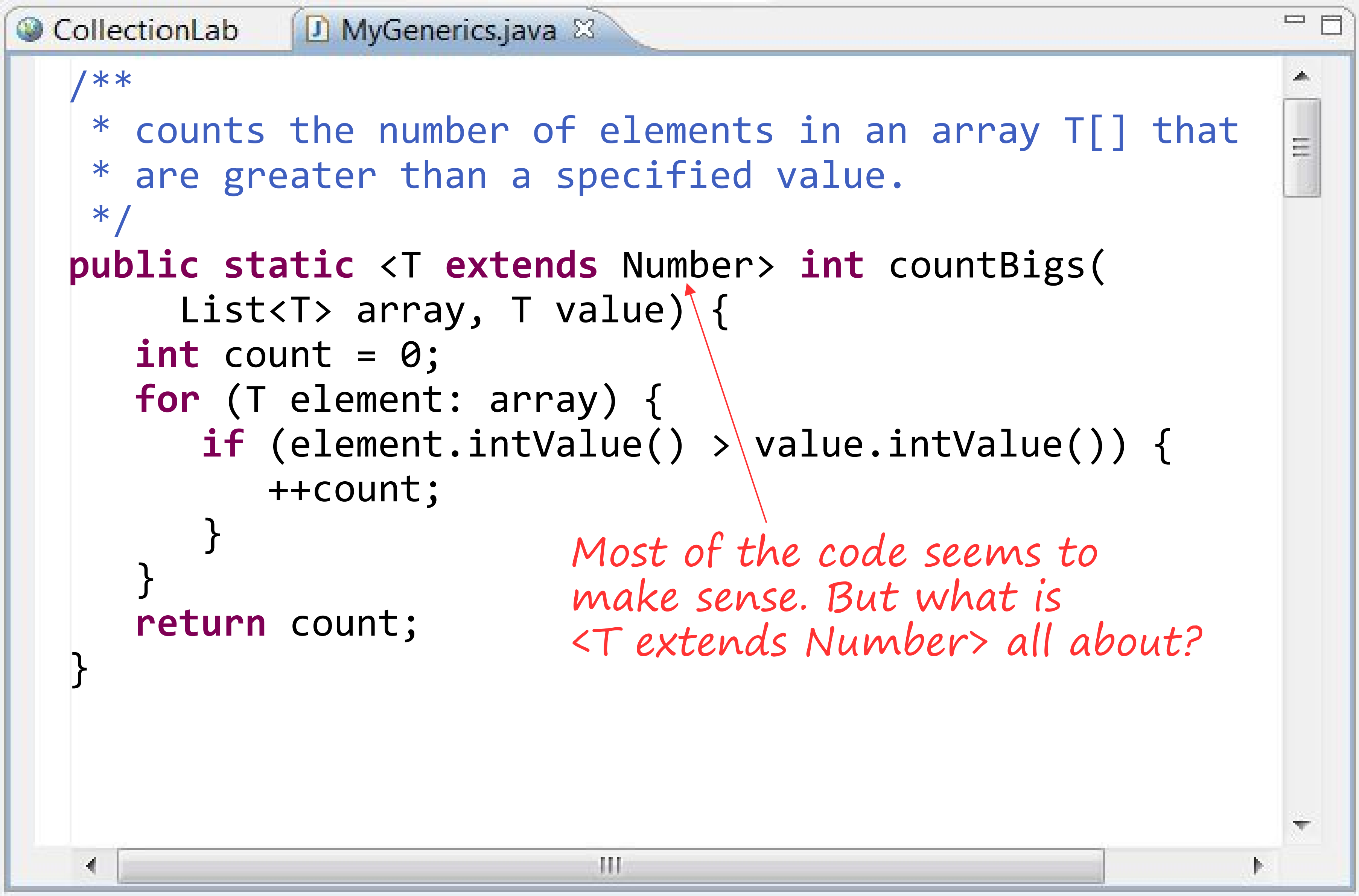
you can define methods that use Generics; often they are static methods.

e.g. suppose we want a method that counts all the elements of a collection that are larger than a specified value.



3 elements > value

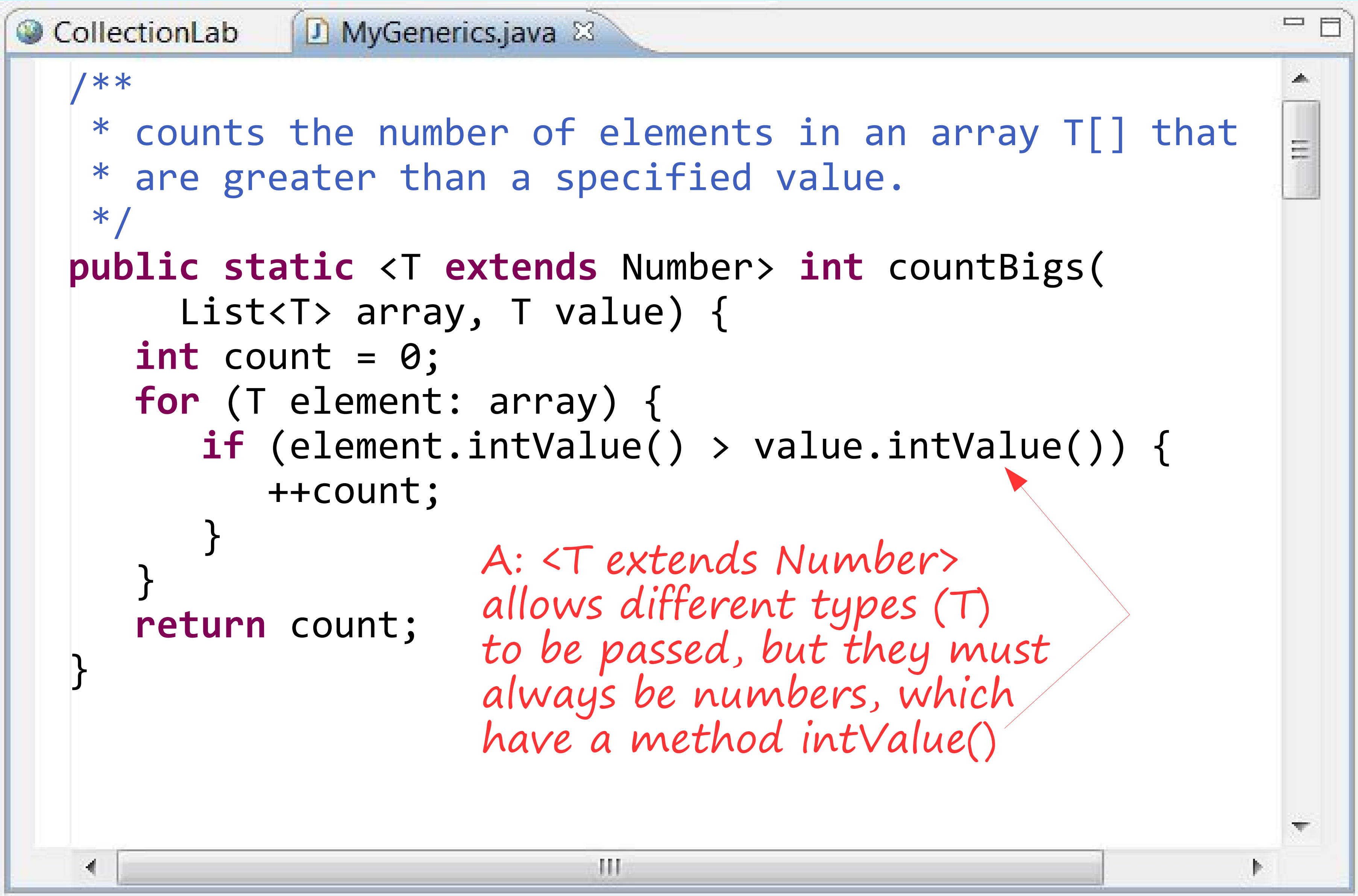
Defining Generic type for method



```
/**
 * counts the number of elements in an array T[] that
 * are greater than a specified value.
 */
public static <T extends Number> int countBigs(
     List<T> array, T value) {
   int count = 0;
   for (T element: array) {
      if (element.intValue() > value.intValue()) {
         ++count;
      }
   }
   return count;
}
```

Most of the code seems to make sense. But what is <T extends Number> all about?

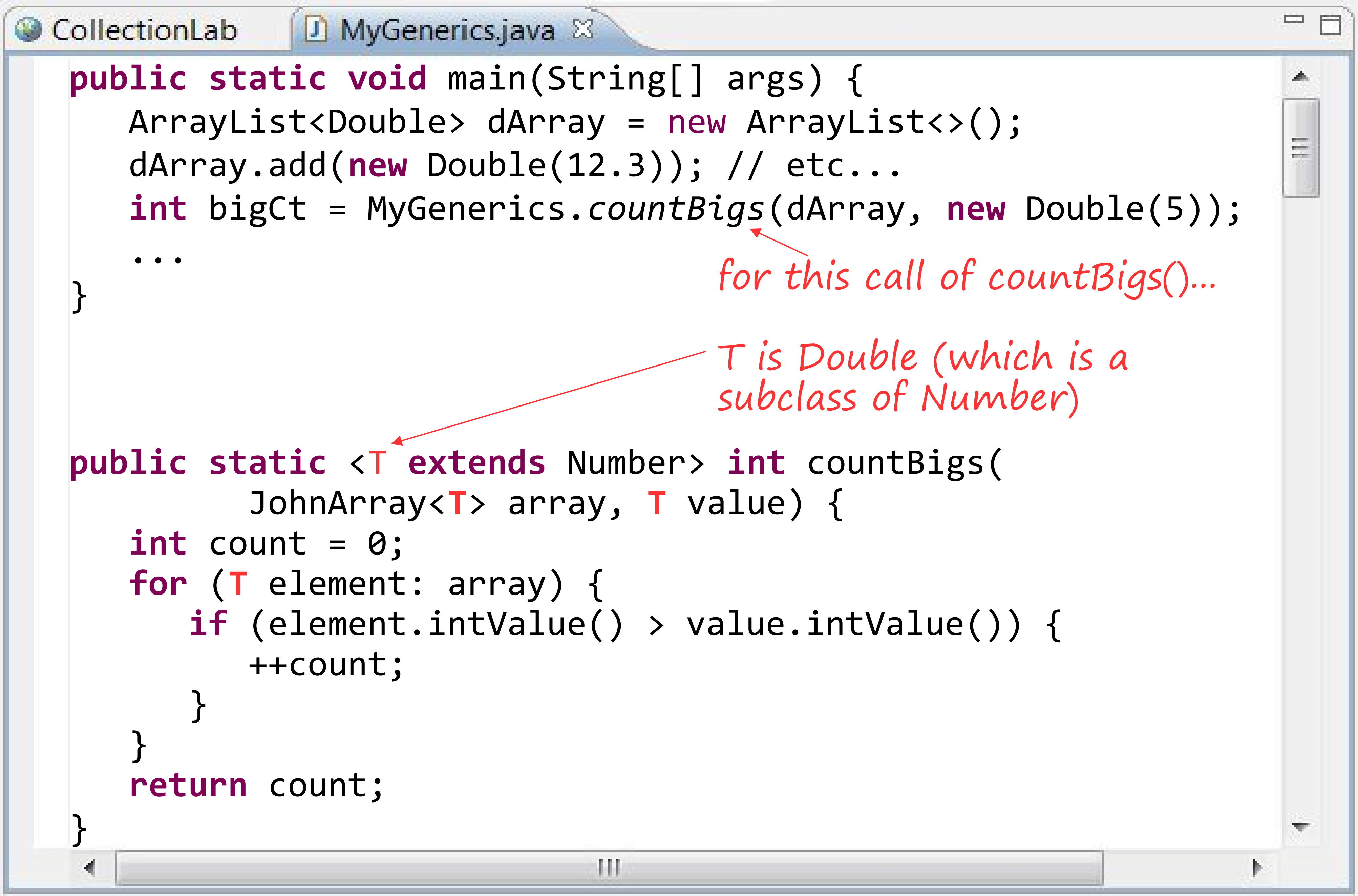
Defining Generic Method - 2



```java
/**
 * counts the number of elements in an array T[] that
 * are greater than a specified value.
 */
public static <T extends Number> int countBigs(
     List<T> array, T value) {
   int count = 0;
   for (T element: array) {
      if (element.intValue() > value.intValue()) {
         ++count;
      }
   }
   return count;
}
```

A: <T extends Number> allows different types (T) to be passed, but they must always be numbers, which have a method intValue()

Invoking Generic Method

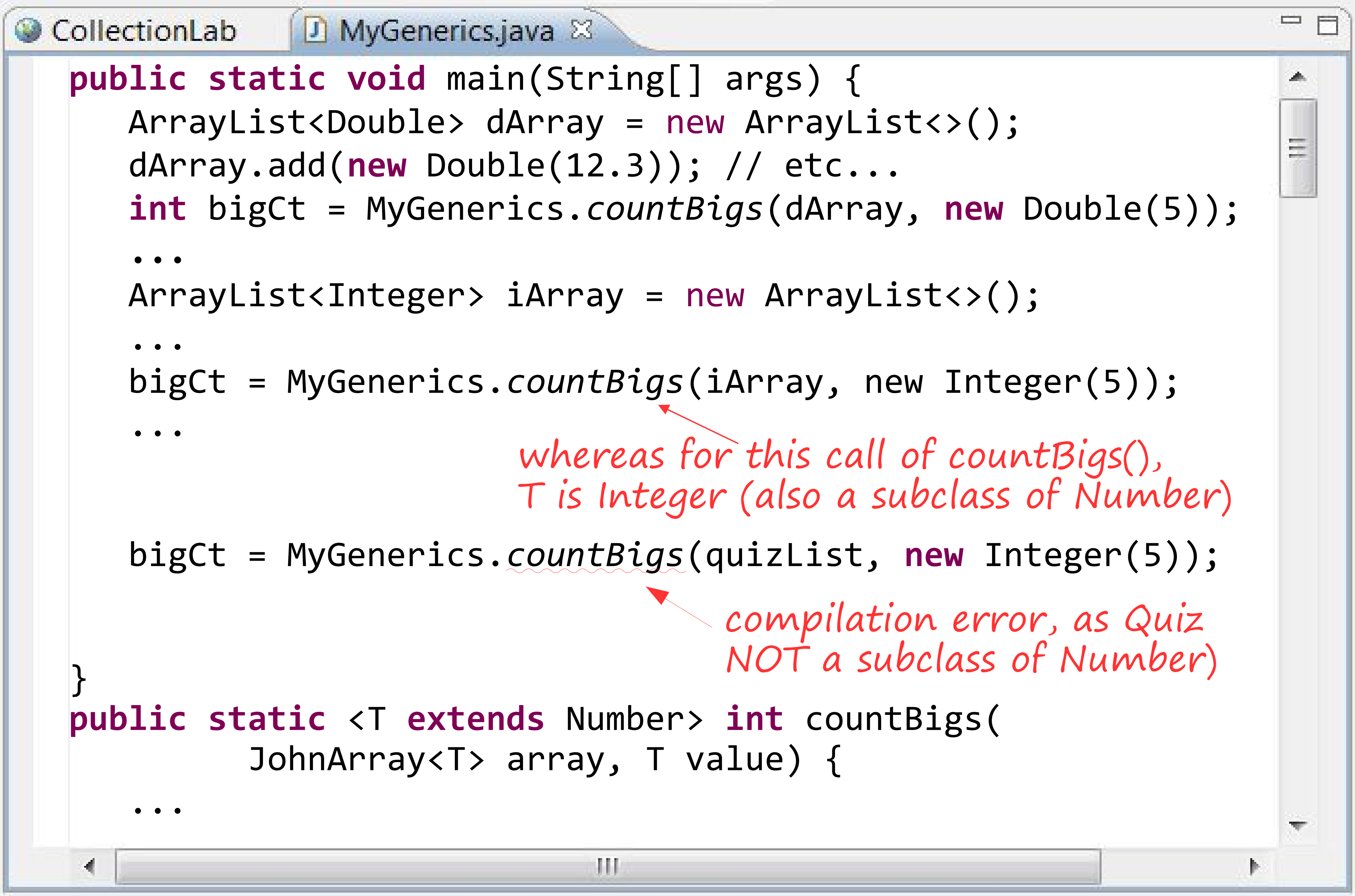


```
public static void main(String[] args) {
   ArrayList<Double> dArray = new ArrayList<>();
   dArray.add(new Double(12.3)); // etc...
   int bigCt = MyGenerics.countBigs(dArray, new Double(5));
   ...
}


public static <T extends Number> int countBigs(
         JohnArray<T> array, T value) {
   int count = 0;
   for (T element: array) {
      if (element.intValue() > value.intValue()) {
         ++count;
      }
   }
   return count;
}
```

for this call of countBigs()...

T is Double (which is a subclass of Number)

Invoking Generic Method - 2



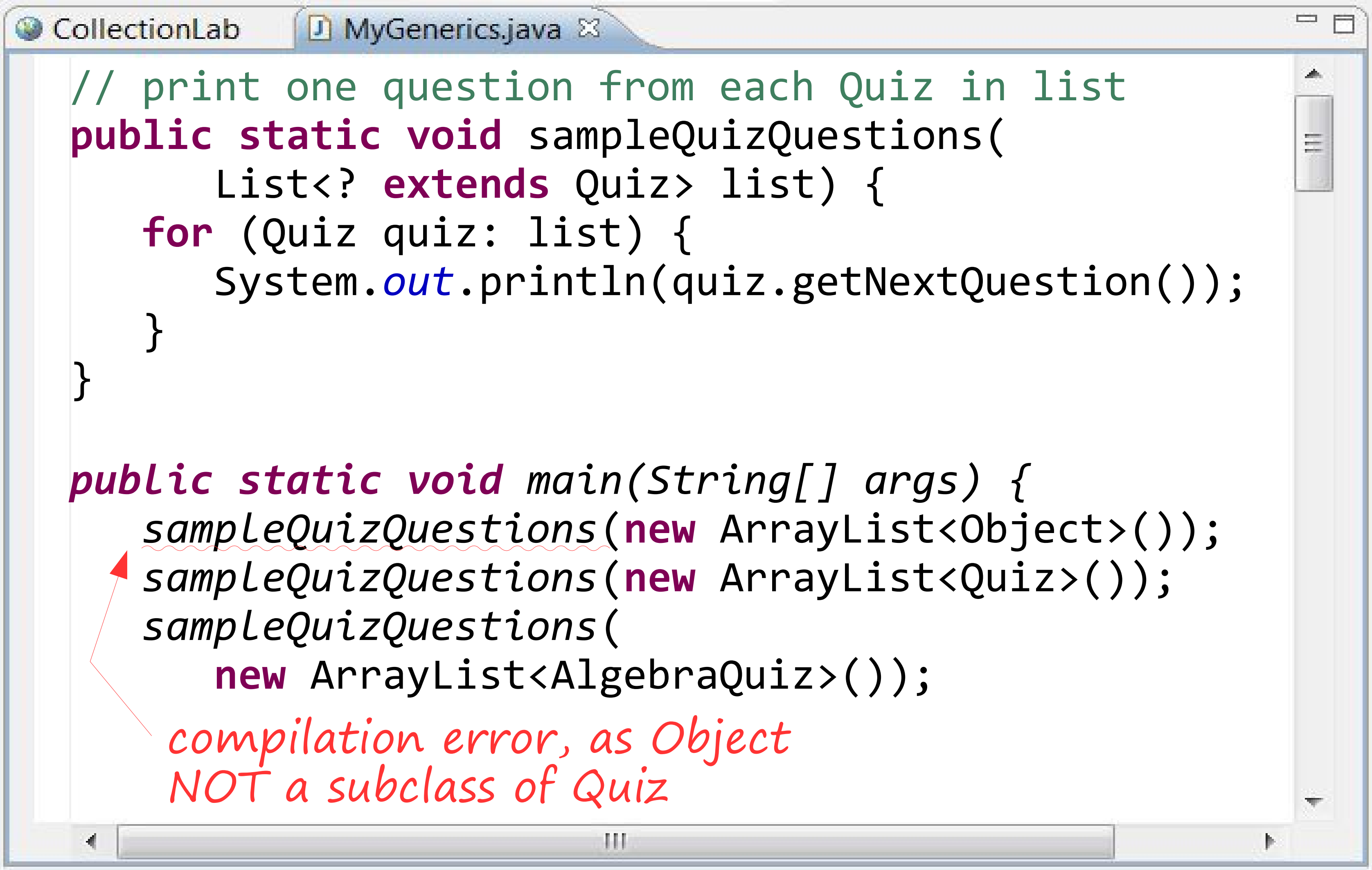
```
public static void main(String[] args) {
   ArrayList<Double> dArray = new ArrayList<>();
   dArray.add(new Double(12.3)); // etc...
   int bigCt = MyGenerics.countBigs(dArray, new Double(5));
   ...
   ArrayList<Integer> iArray = new ArrayList<>();
   ...
   bigCt = MyGenerics.countBigs(iArray, new Integer(5));
   ...

   bigCt = MyGenerics.countBigs(quizList, new Integer(5));

}
public static <T extends Number> int countBigs(
         JohnArray<T> array, T value) {
   ...
```

whereas for this call of countBigs(),
T is Integer (also a subclass of Number)

compilation error, as Quiz
NOT a subclass of Number)

Wildcards – Example 1

The upper bounded wildcard, e.g.
<? **extends** Quiz> is used when we want
to take values out of the parameter.

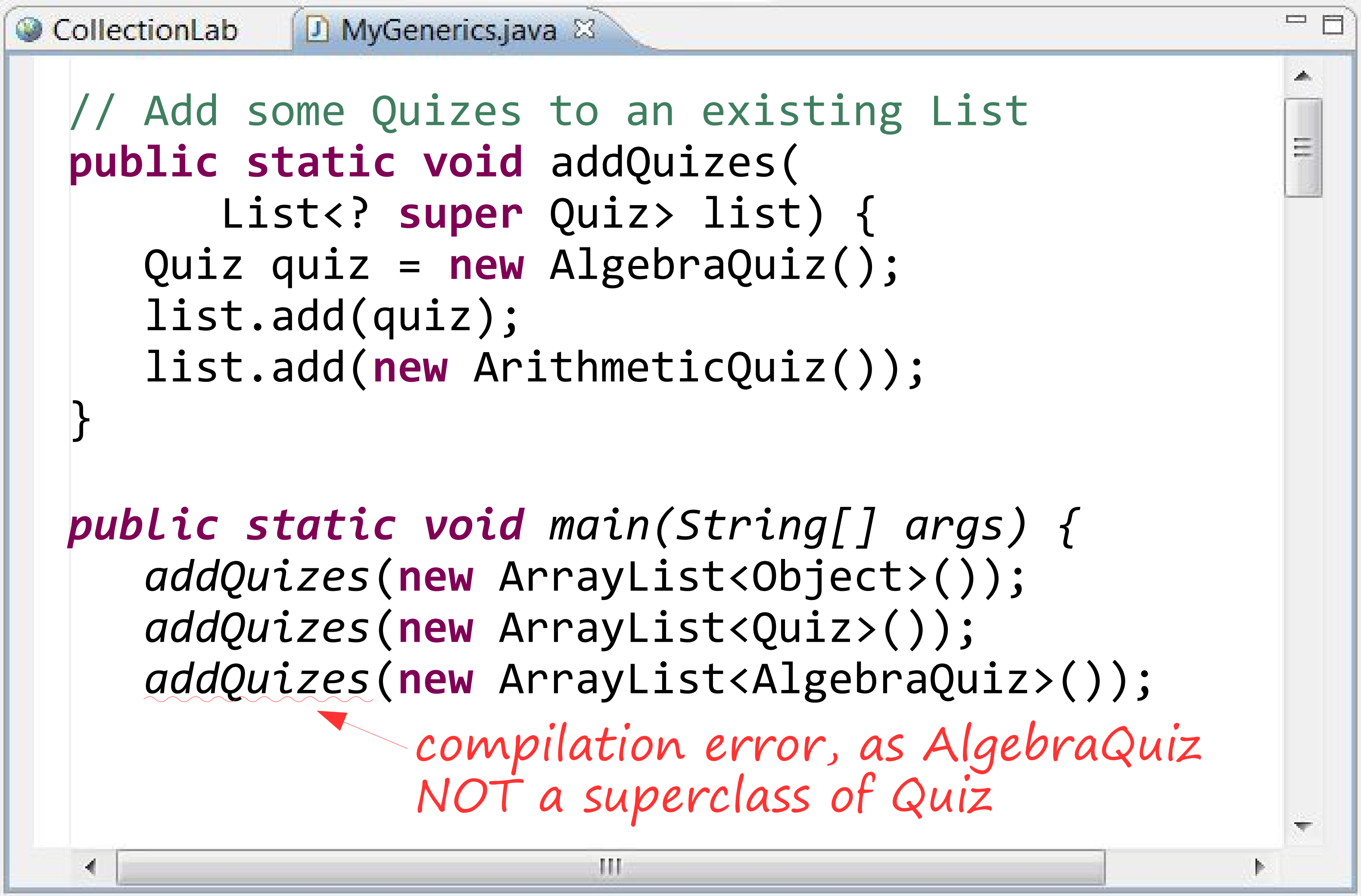


```
// print one question from each Quiz in list
public static void sampleQuizQuestions(
        List<? extends Quiz> list) {
    for (Quiz quiz: list) {
        System.out.println(quiz.getNextQuestion());
    }
}

public static void main(String[] args) {
    sampleQuizQuestions(new ArrayList<Object>());
    sampleQuizQuestions(new ArrayList<Quiz>());
    sampleQuizQuestions(
        new ArrayList<AlgebraQuiz>());
```

compilation error, as Object
NOT a subclass of Quiz

Wildcards – Example 2

Lower bounded wildcard, e.g.
<? **super** Quiz> is when we want to
add values to the parameter.

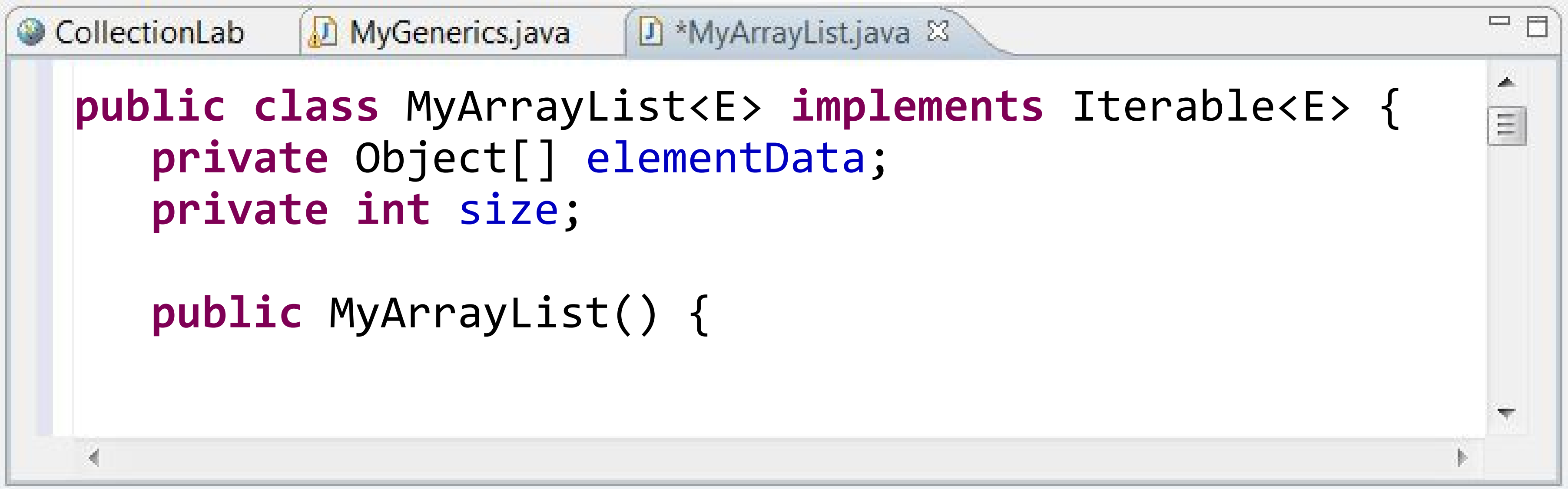


```
// Add some Quizes to an existing List
public static void addQuizes(
      List<? super Quiz> list) {
   Quiz quiz = new AlgebraQuiz();
   list.add(quiz);
   list.add(new ArithmeticQuiz());
}

public static void main(String[] args) {
   addQuizes(new ArrayList<Object>());
   addQuizes(new ArrayList<Quiz>());
   addQuizes(new ArrayList<AlgebraQuiz>());
```

compilation error, as AlgebraQuiz
NOT a superclass of Quiz

Defining Generic Collections

You can define your own Generic collection classes. **demo.generics.MyArrayList** is a simplified version of java.util.ArrayList<E> to illustrate some of the ideas involved, e.g. iterating and sorting the data.



```
public class MyArrayList<E> implements Iterable<E> {
   private Object[] elementData;
   private int size;

   public MyArrayList() {
```

Why is `elementData` defined as `Object[]` and not as `T[]`? See Notes.

Generics – Summary

- added at Java 5
- provide compile-time type safety
- reduce risk of programmer errors
- more explicit description of code
- mostly used with Collections

Further reading: http://docs.oracle.com/javase/tutorial/java/generics/

Exercise: Collections and Generics