Generics

Generic Programming in Java

Outline

- History and theory
- Generic type parameter
- Generic Interface , Class and method
- Bounded type parameter
- Inheritance with generic classes
- Overriding with generic classes
- Restriction and Limitations
- Conclusion

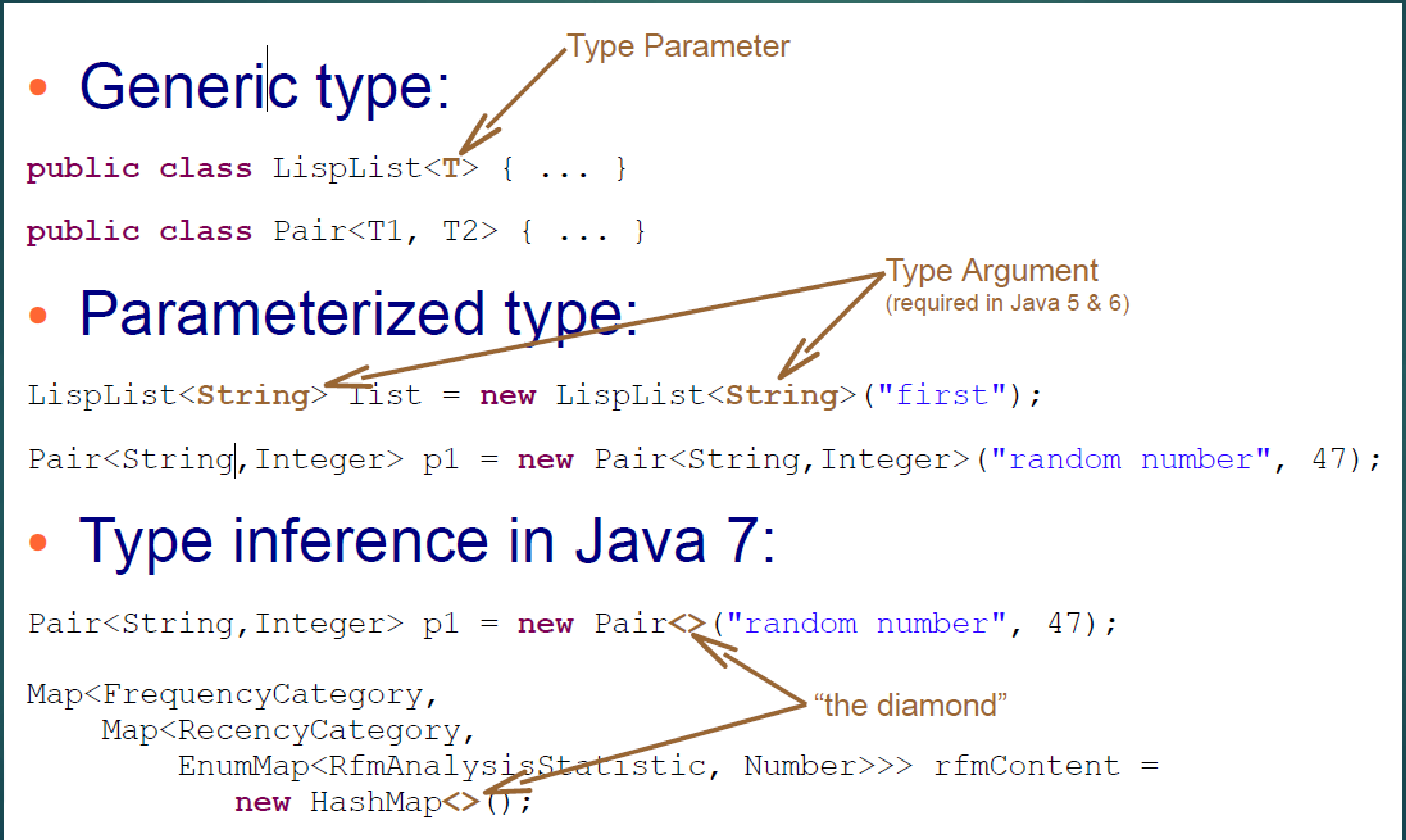
History

- Generic types were introduced in J2SE 5.0 in 2004
  - Additional type safety
  - Reduced the need for casting
- Pre-generics code example:
  - List v = new ArrayList();
  - v.add(“test”);
  - Integer i = (Integer) v.get(0); // Run-time error
- Post-generics:
  - List<String> v = new ArrayList<String>();
  - v.add(“test”);
  - Integer I = v.get(0); // Compile-time error

Why Generics method ?

- Generic methods allow you to create algorithms that apply to a wide variety of types.
- For example, how many sorting algorithms do you need ?
  - Sort a list of integers
  - Sort a list of dates
  - Sort a list of strings
- What do these all have in common ?
- "Get your data structures correct first, and the rest of the program will write itself."
  - *David Jones*

Terminology



Type Parameter
Type Argument
(required in Java 5 & 6)
“the diamond”

- Generic type:

```
public class LispList<T> { ... }

public class Pair<T1, T2> { ... }
```

- Parameterized type:

```
LispList<String> list = new LispList<String>("first");

Pair<String,Integer> p1 = new Pair<String,Integer>("random number", 47);
```

- Type inference in Java 7:

```
Pair<String,Integer> p1 = new Pair<>("random number", 47);

Map<FrequencyCategory,
    Map<RecencyCategory,
        EnumMap<RfmAnalysisStatistic, Number>>> rfmContent =
           new HashMap<>();
```

Theory

- Primitives cannot be type parameters
  - List<int> numbers; // illegal
- Generics are a compile-time feature
  - At run-time type variables (ex. T become Object)
  - This is called erasure
  - Public class LispList<T> {
    - private T item;
    - private LispList<T> next;
    - public T first() {…}
    - // etc…
  - }

→

Public class LispList {

    private Object item;

    private LispList next;

    public Object first() {…}

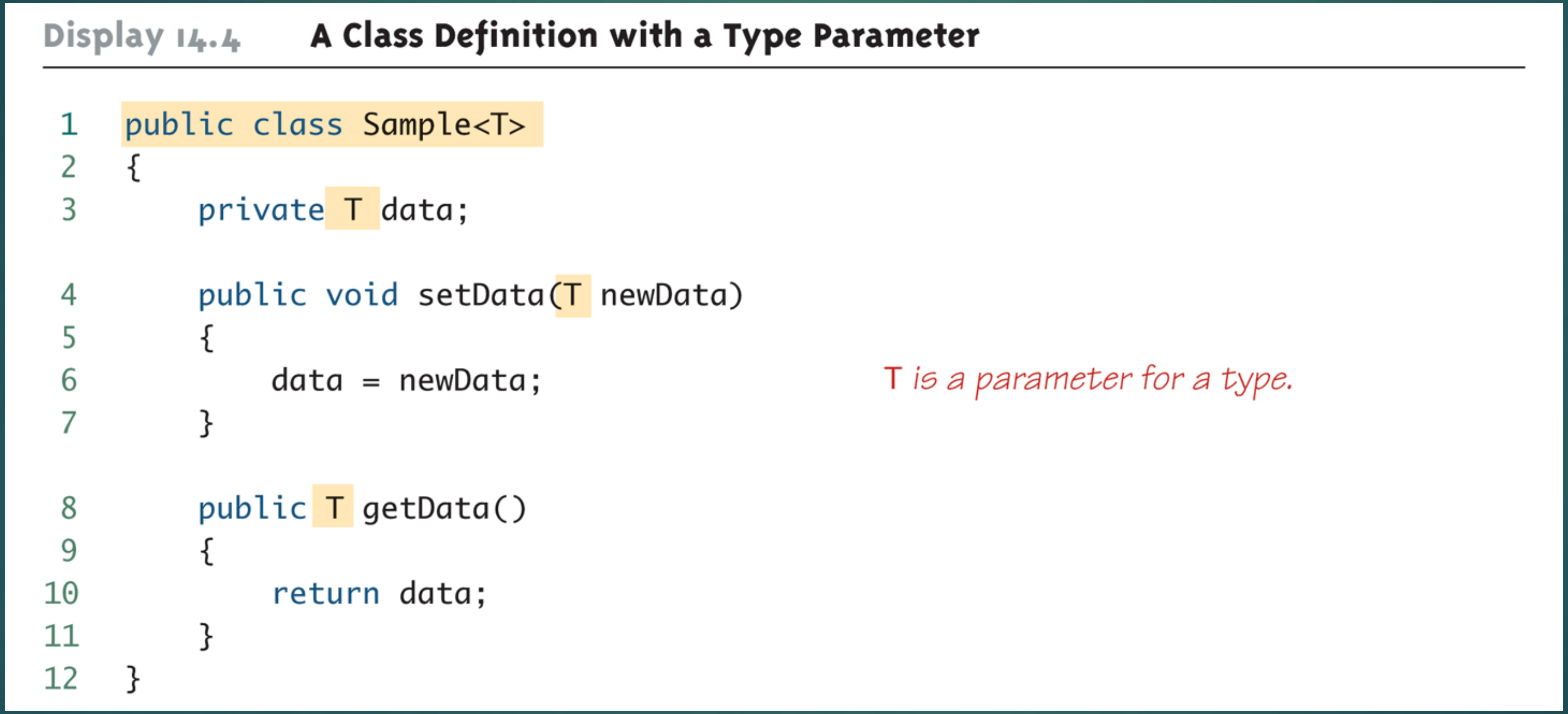
    // etc…

}

Generics type parameter

- Classes and methods can have a type parameter
  - A type parameter can have any reference type (i.e., any class type) plugged in for the type parameter. (T → Object and any subclasses)
  - When a specific type is plugged in, this produces a specific class type or method
    (ex. If We use T as Integer then String class are not allowed)
  - Traditionally, a single uppercase letter is used for a type parameter, but any non-keyword identifier may be used.

A class definition with a type parameter

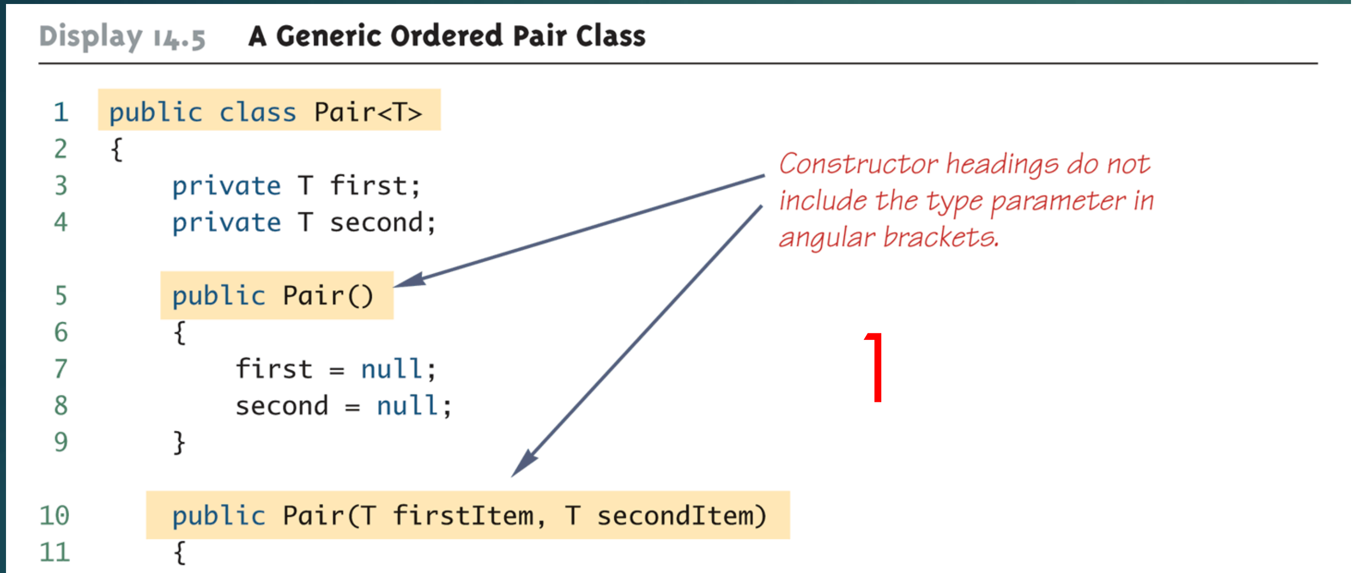


Display 14.4 **A Class Definition with a Type Parameter**

```java
public class Sample<T>
{
    private T data;

    public void setData(T newData)
    {
        data = newData;
    }

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

T is a parameter for a type.

A Generic Pair Class

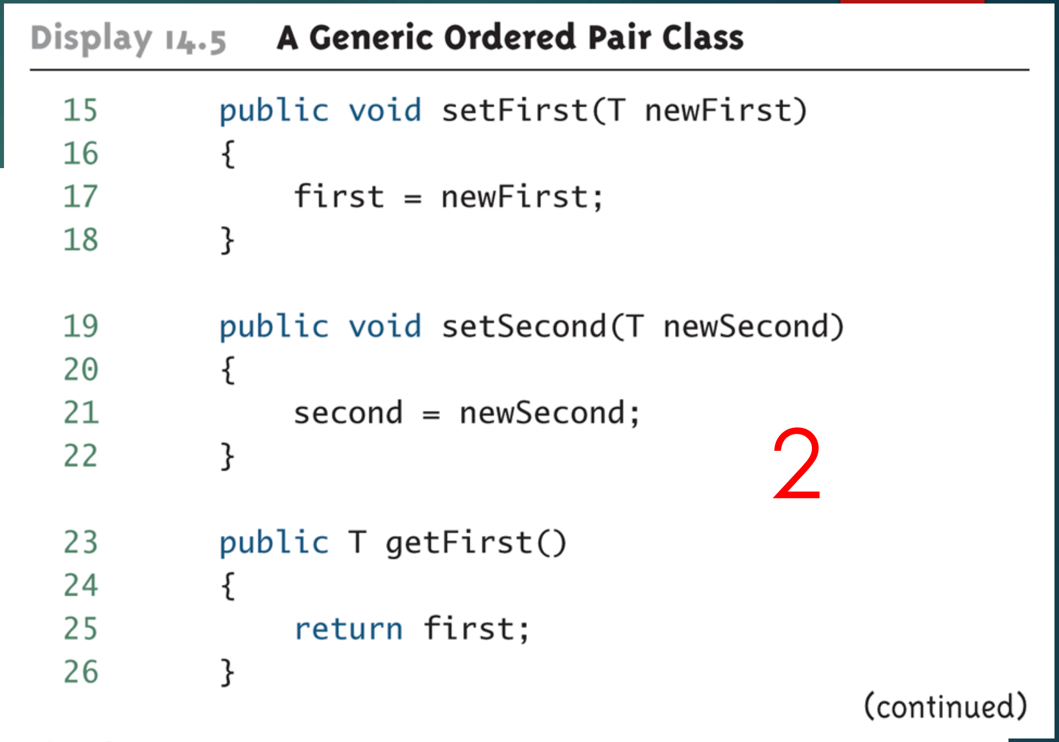


Display 14.5 **A Generic Ordered Pair Class**

```
public class Pair<T>
{
    private T first;
    private T second;

    public Pair()
    {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
```

Constructor headings do not include the type parameter in angular brackets.

1

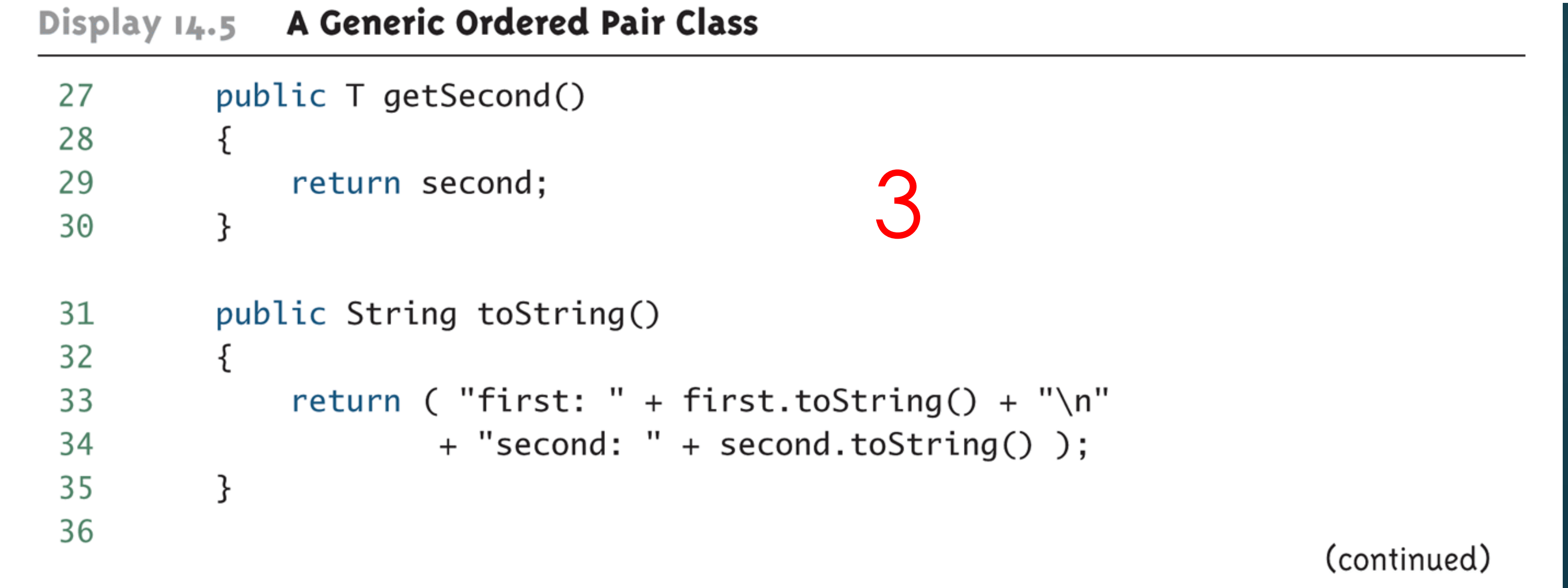


Display 14.5 **A Generic Ordered Pair Class**

```
    public void setFirst(T newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T newSecond)
    {
        second = newSecond;
    }

    public T getFirst()
    {
        return first;
    }
```

(continued)

2



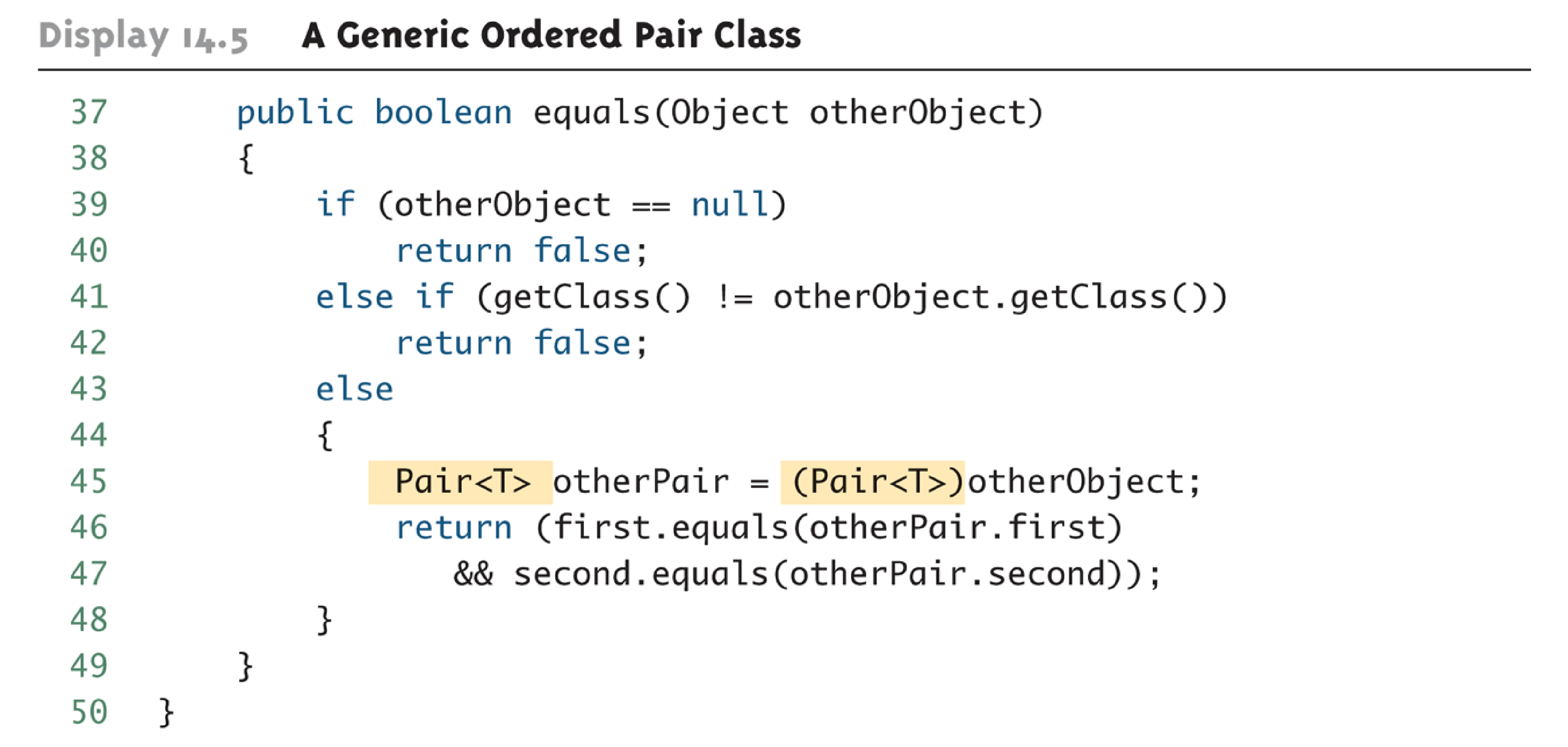
Display 14.5 **A Generic Ordered Pair Class**

```
    public T getSecond()
    {
        return second;
    }

    public String toString()
    {
        return ( "first: " + first.toString() + "\n"
                 + "second: " + second.toString() );
    }

```

(continued)

3

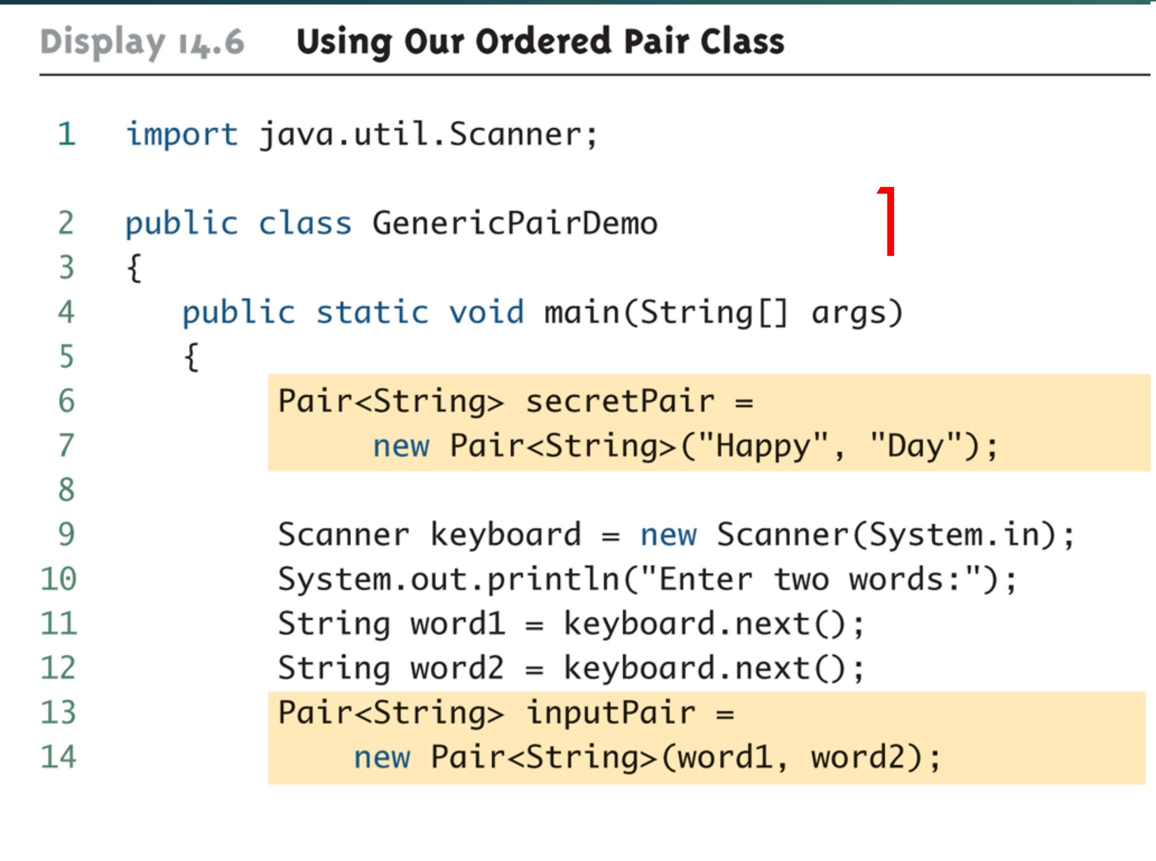
A Generic Pair Class (cont’d)



Display 14.5 A Generic Ordered Pair Class

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            Pair<T> otherPair = (Pair<T>)otherObject;
            return (first.equals(otherPair.first)
               && second.equals(otherPair.second));
        }
    }
}
```

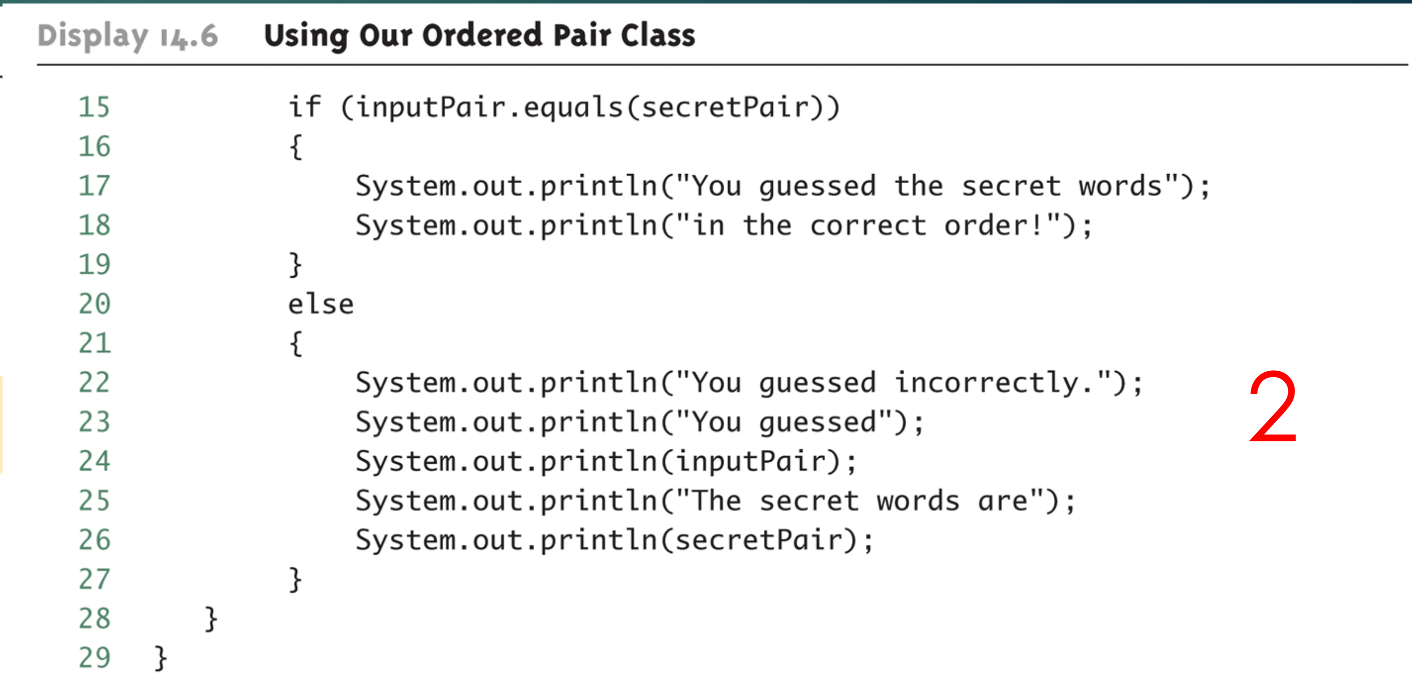
A Generic Pair Class (cont’d)



Display 14.6 **Using Our Ordered Pair Class**

```java
import java.util.Scanner;

public class GenericPairDemo
{
    public static void main(String[] args)
    {
        Pair<String> secretPair =
             new Pair<String>("Happy", "Day");

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two words:");
        String word1 = keyboard.next();
        String word2 = keyboard.next();
        Pair<String> inputPair =
            new Pair<String>(word1, word2);
```

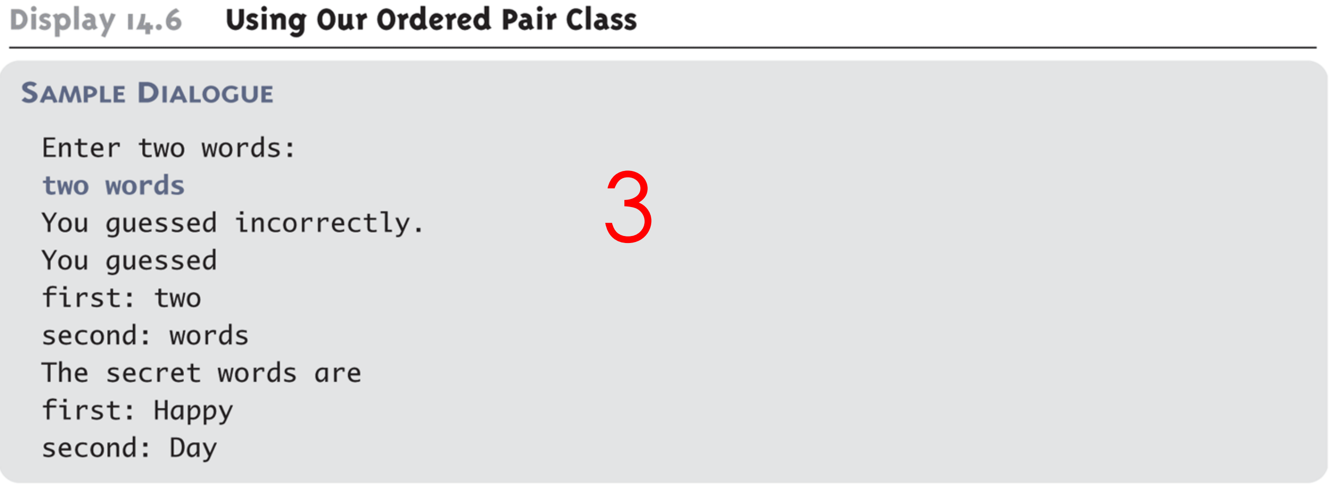
1



Display 14.6 **Using Our Ordered Pair Class**

```java
        if (inputPair.equals(secretPair))
        {
            System.out.println("You guessed the secret words");
            System.out.println("in the correct order!");
        }
        else
        {
            System.out.println("You guessed incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret words are");
            System.out.println(secretPair);
        }
    }
}
```

2



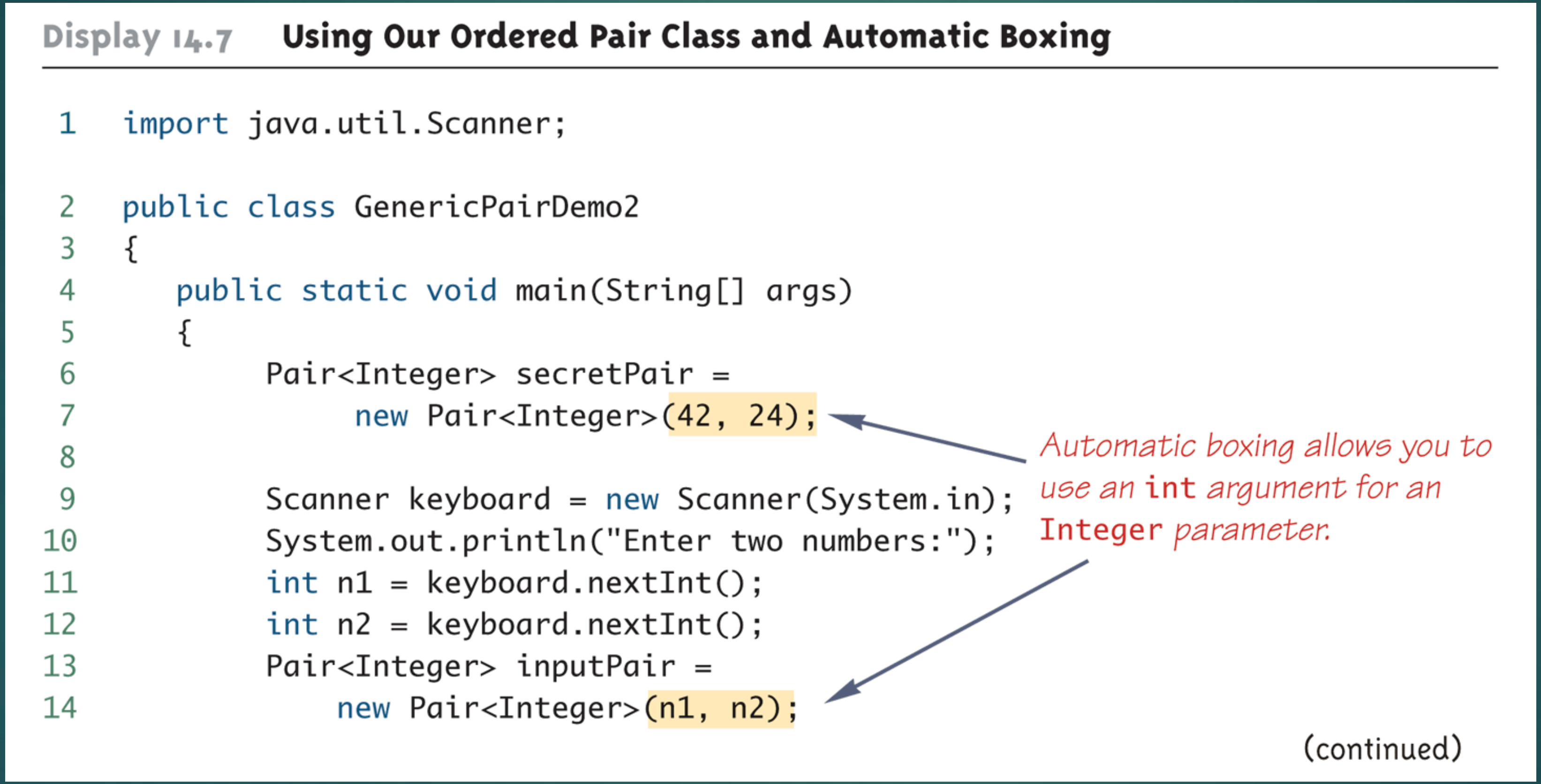
Display 14.6 **Using Our Ordered Pair Class**

Sample Dialogue

```
Enter two words:
two words
You guessed incorrectly.
You guessed
first: two
second: words
The secret words are
first: Happy
second: Day
```

3

Generic Pair class and Automatic Boxing



Display 14.7 **Using Our Ordered Pair Class and Automatic Boxing**

```java
import java.util.Scanner;

public class GenericPairDemo2
{
   public static void main(String[] args)
   {
        Pair<Integer> secretPair =
             new Pair<Integer>(42, 24);

        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter two numbers:");
        int n1 = keyboard.nextInt();
        int n2 = keyboard.nextInt();
        Pair<Integer> inputPair =
            new Pair<Integer>(n1, n2);
```

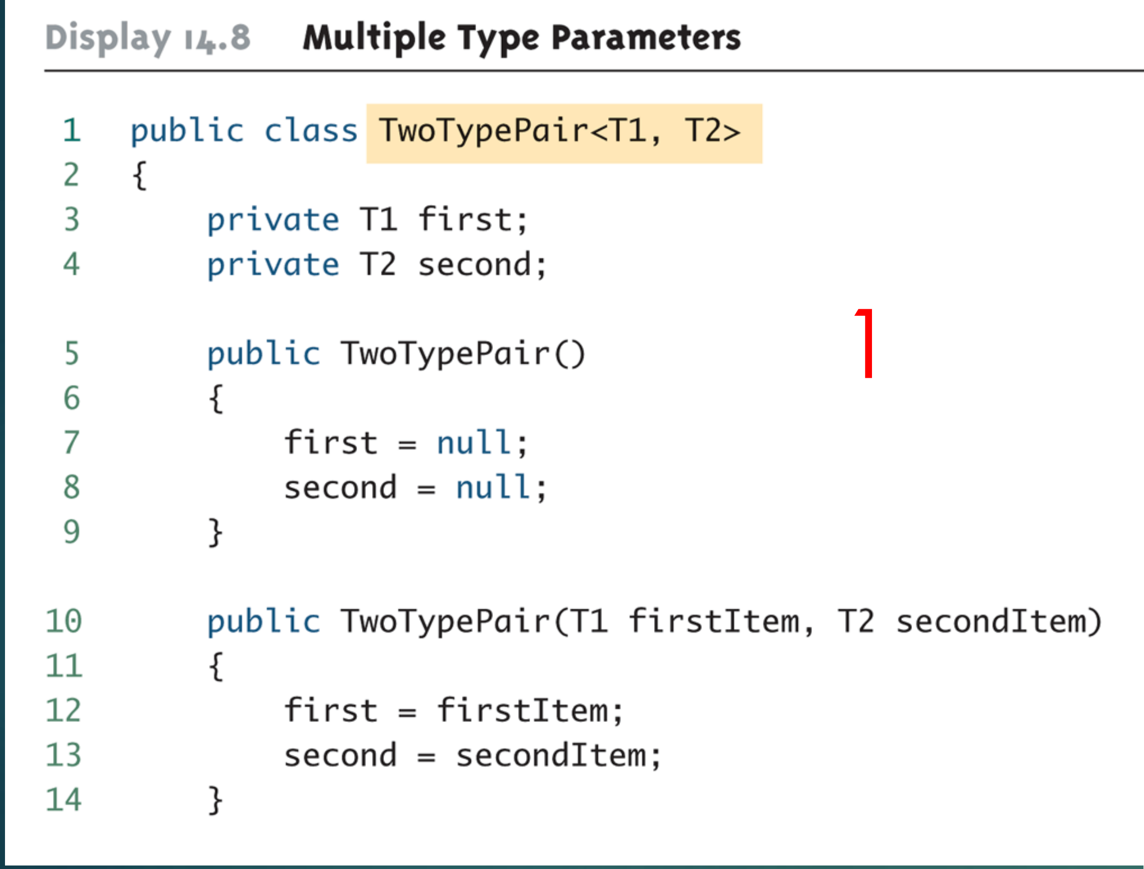
Automatic boxing allows you to use an **int** *argument for an* **Integer** *parameter.*

(continued)

A Class Definition Can Have More Than One Type Parameter

- A generic class definition can have any number of type parameters
  - Multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas

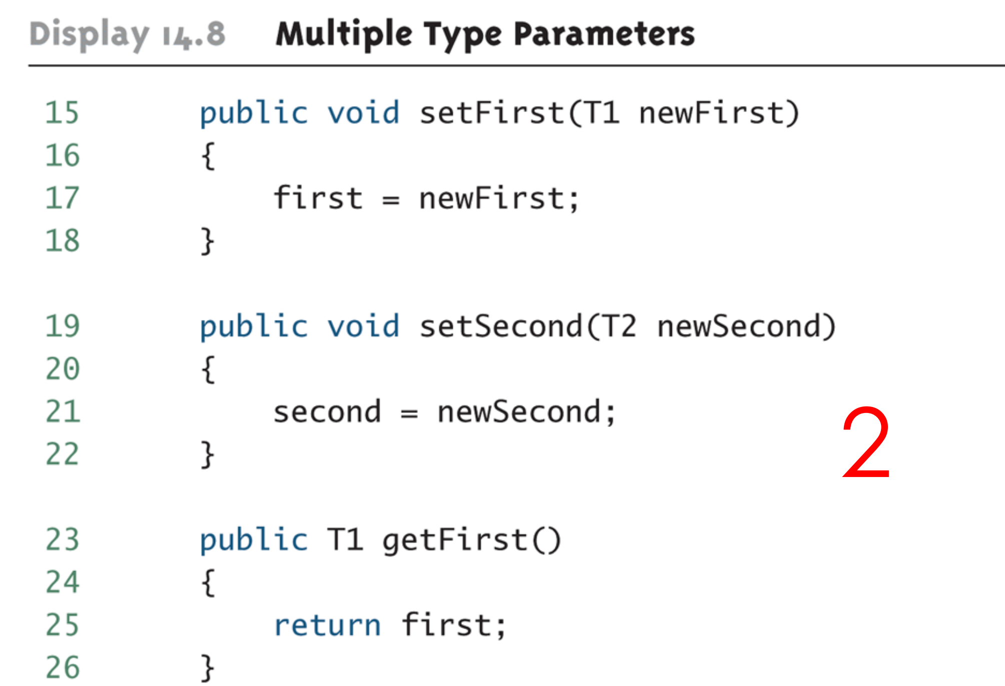
Multiple Type Parameters



Display 14.8 Multiple Type Parameters

```java
public class TwoTypePair<T1, T2>
{
    private T1 first;
    private T2 second;

    public TwoTypePair()
    {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem)
    {
        first = firstItem;
        second = secondItem;
    }
```

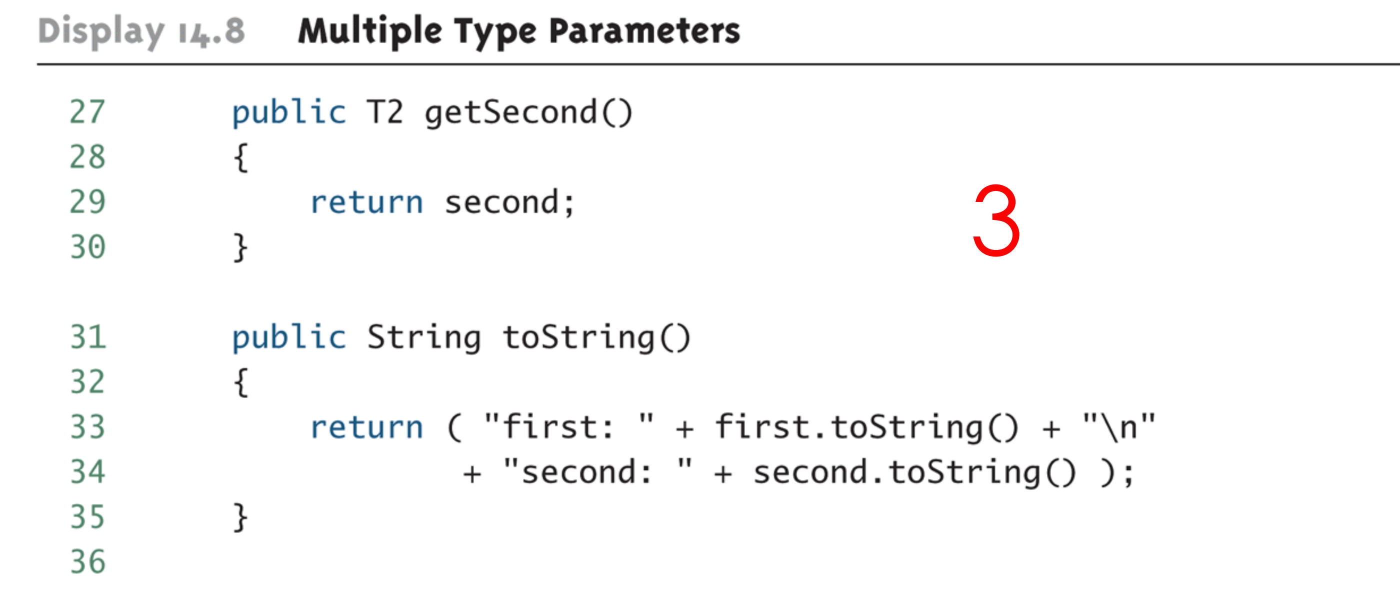
1



Display 14.8 Multiple Type Parameters

```java
    public void setFirst(T1 newFirst)
    {
        first = newFirst;
    }

    public void setSecond(T2 newSecond)
    {
        second = newSecond;
    }

    public T1 getFirst()
    {
        return first;
    }
```

2

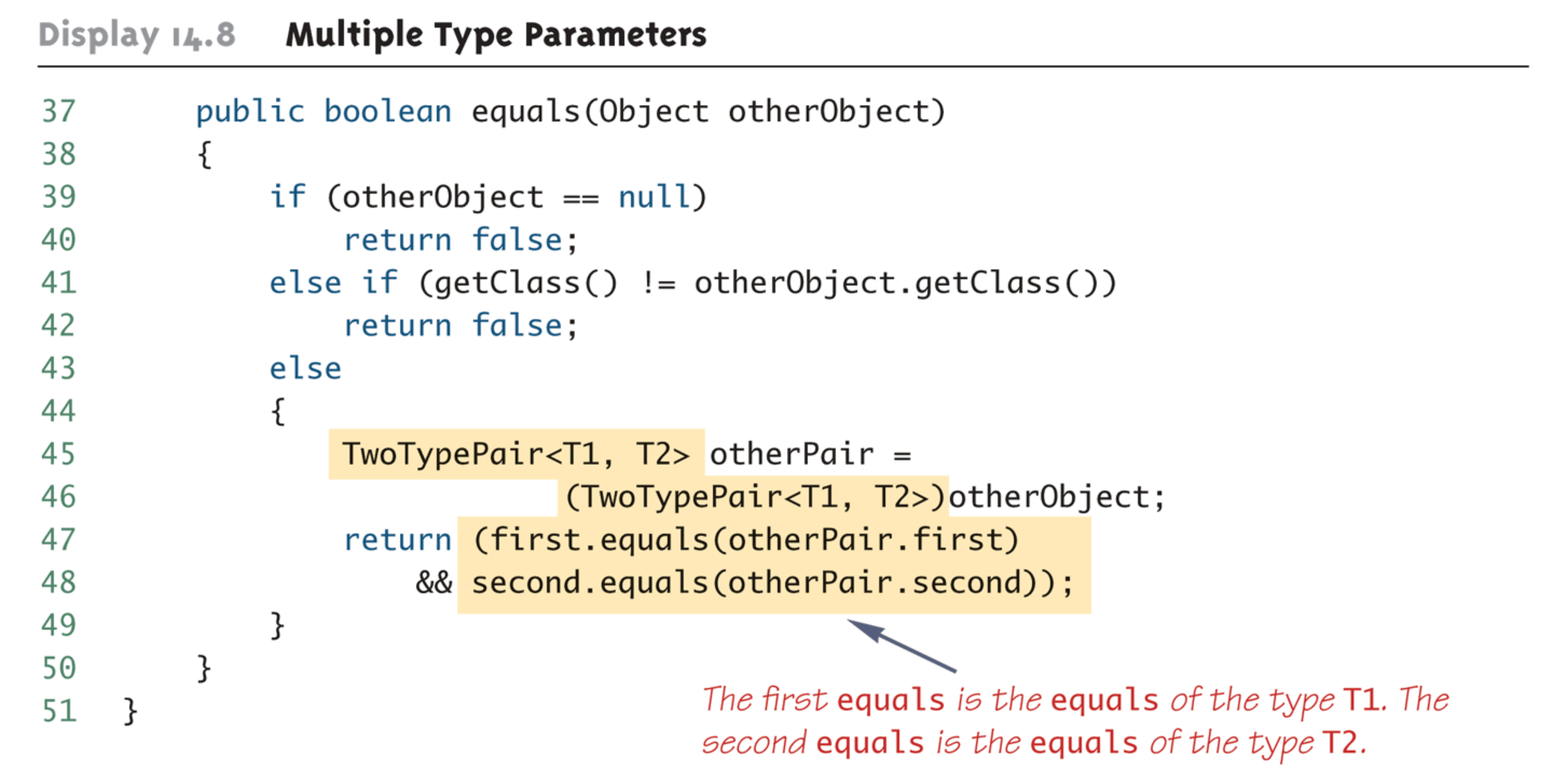


Display 14.8 Multiple Type Parameters

```java
    public T2 getSecond()
    {
        return second;
    }

    public String toString()
    {
        return ( "first: " + first.toString() + "\n"
                 + "second: " + second.toString() );
    }

```

3

Multiple Type Parameters (cont’d)

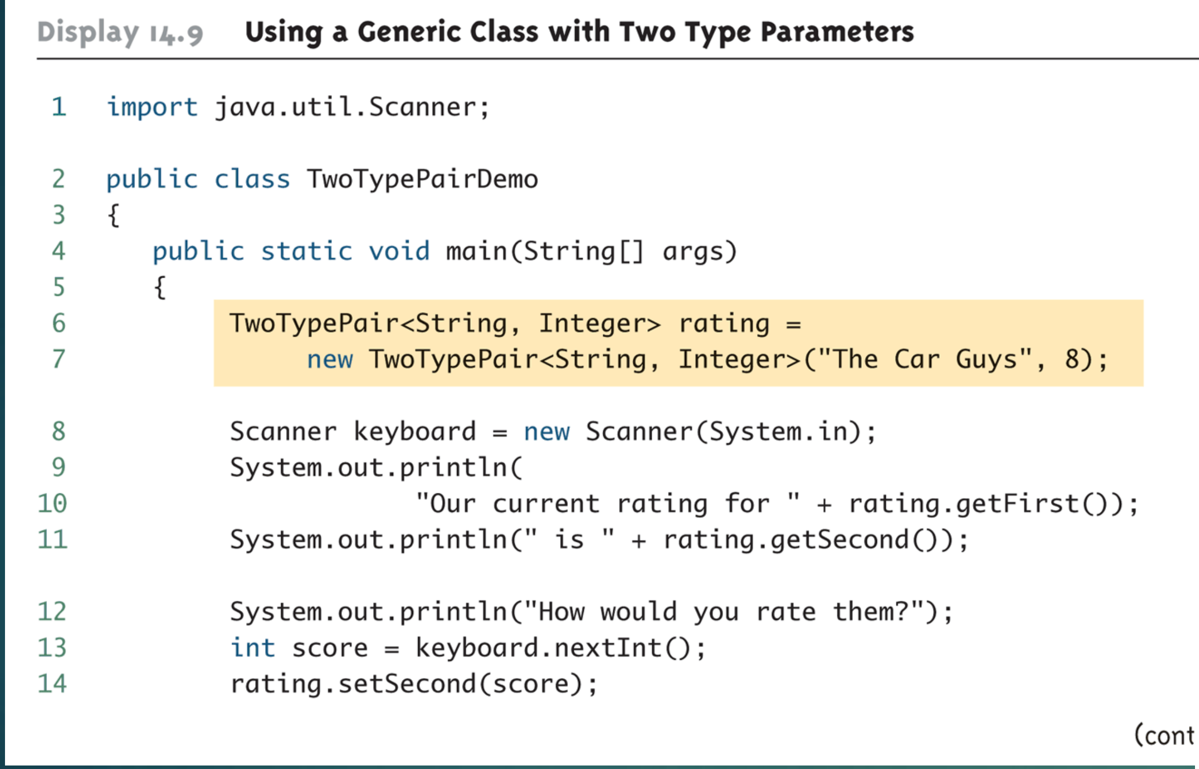


Display 14.8 Multiple Type Parameters

```
    public boolean equals(Object otherObject)
    {
        if (otherObject == null)
            return false;
        else if (getClass() != otherObject.getClass())
            return false;
        else
        {
            TwoTypePair<T1, T2> otherPair =
                        (TwoTypePair<T1, T2>)otherObject;
            return (first.equals(otherPair.first)
                && second.equals(otherPair.second));
        }
    }
}
```

The first **equals** *is the* **equals** *of the type* **T1**. *The second* **equals** *is the* **equals** *of the type* **T2**.

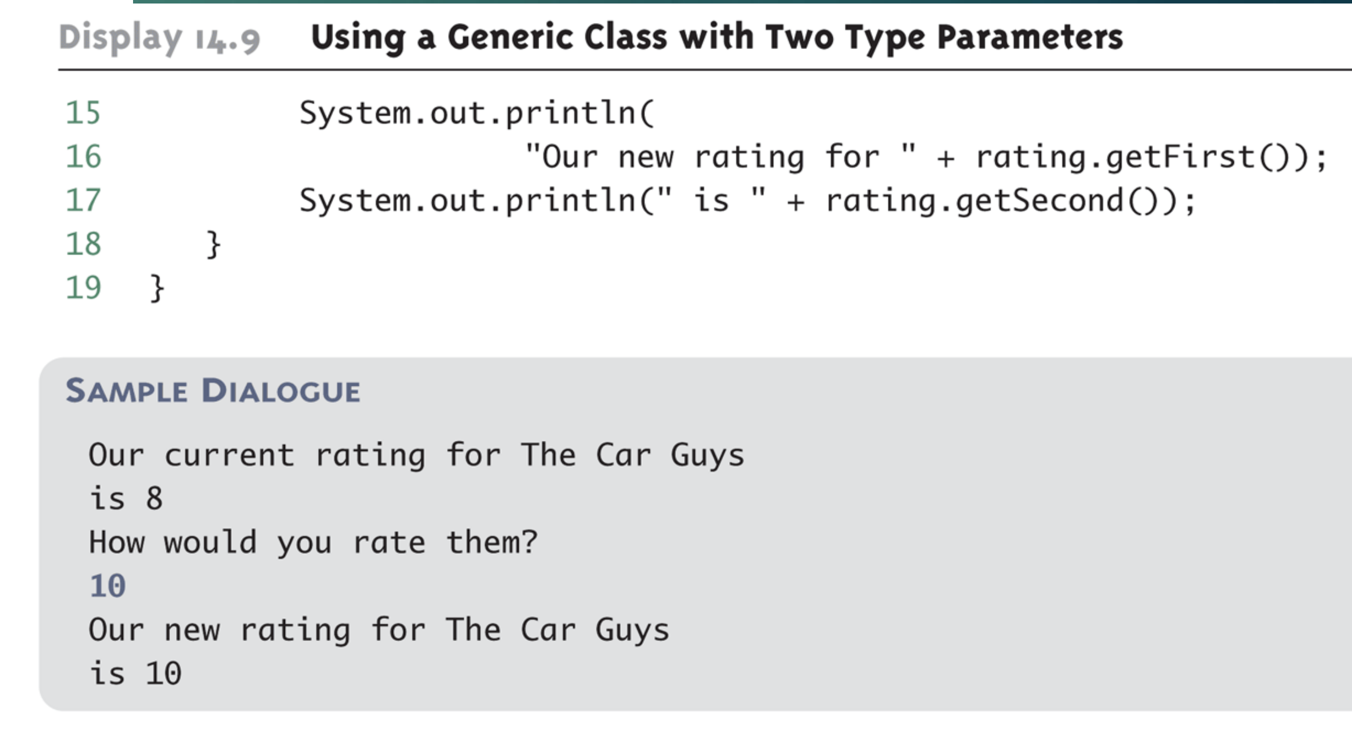
Multiple Type Parameters (cont’d)



Display 14.9 **Using a Generic Class with Two Type Parameters**

```
import java.util.Scanner;

public class TwoTypePairDemo
{
    public static void main(String[] args)
    {
        TwoTypePair<String, Integer> rating =
              new TwoTypePair<String, Integer>("The Car Guys", 8);

        Scanner keyboard = new Scanner(System.in);
        System.out.println(
                    "Our current rating for " + rating.getFirst());
        System.out.println(" is " + rating.getSecond());

        System.out.println("How would you rate them?");
        int score = keyboard.nextInt();
        rating.setSecond(score);
```

(cont



Display 14.9 **Using a Generic Class with Two Type Parameters**

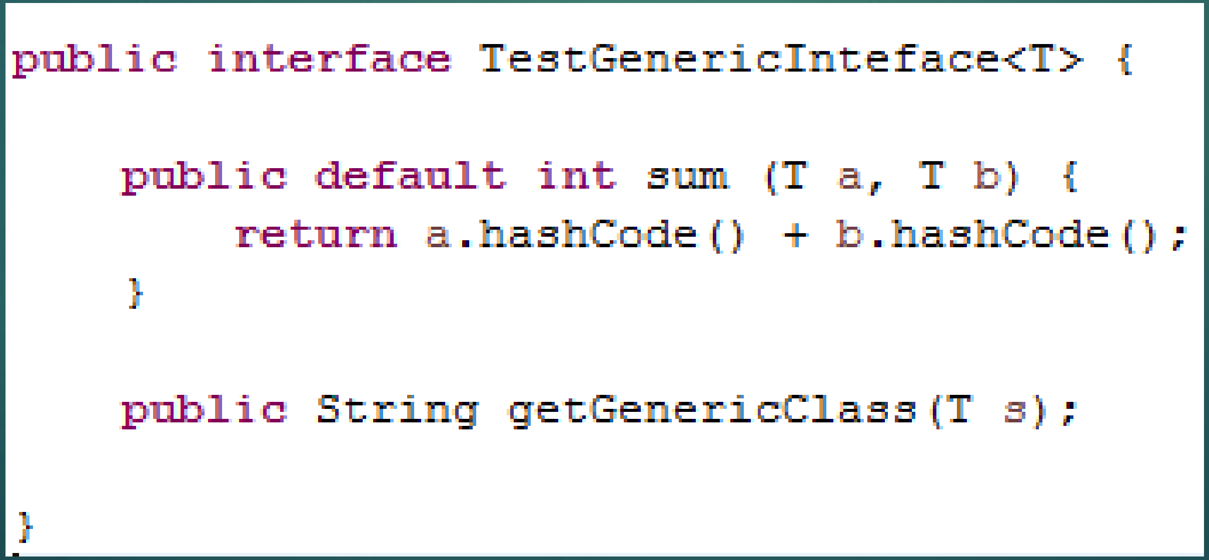
```
        System.out.println(
                    "Our new rating for " + rating.getFirst());
        System.out.println(" is " + rating.getSecond());
    }
}
```

Sample Dialogue

```
Our current rating for The Car Guys
is 8
How would you rate them?
10
Our new rating for The Car Guys
is 10
```

Generic Interfaces

- An interface can have one or more type parameters
- The details and notation are the same as they are for classes with type parameters



```
public interface TestGenericInteface<T> {

    public default int sum (T a, T b) {
        return a.hashCode() + b.hashCode();
    }

    public String getGenericClass(T s);

}
```

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class
  - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter
  - The type parameter of a generic method is local to that method, not to the class

Generic Methods (cont'd)

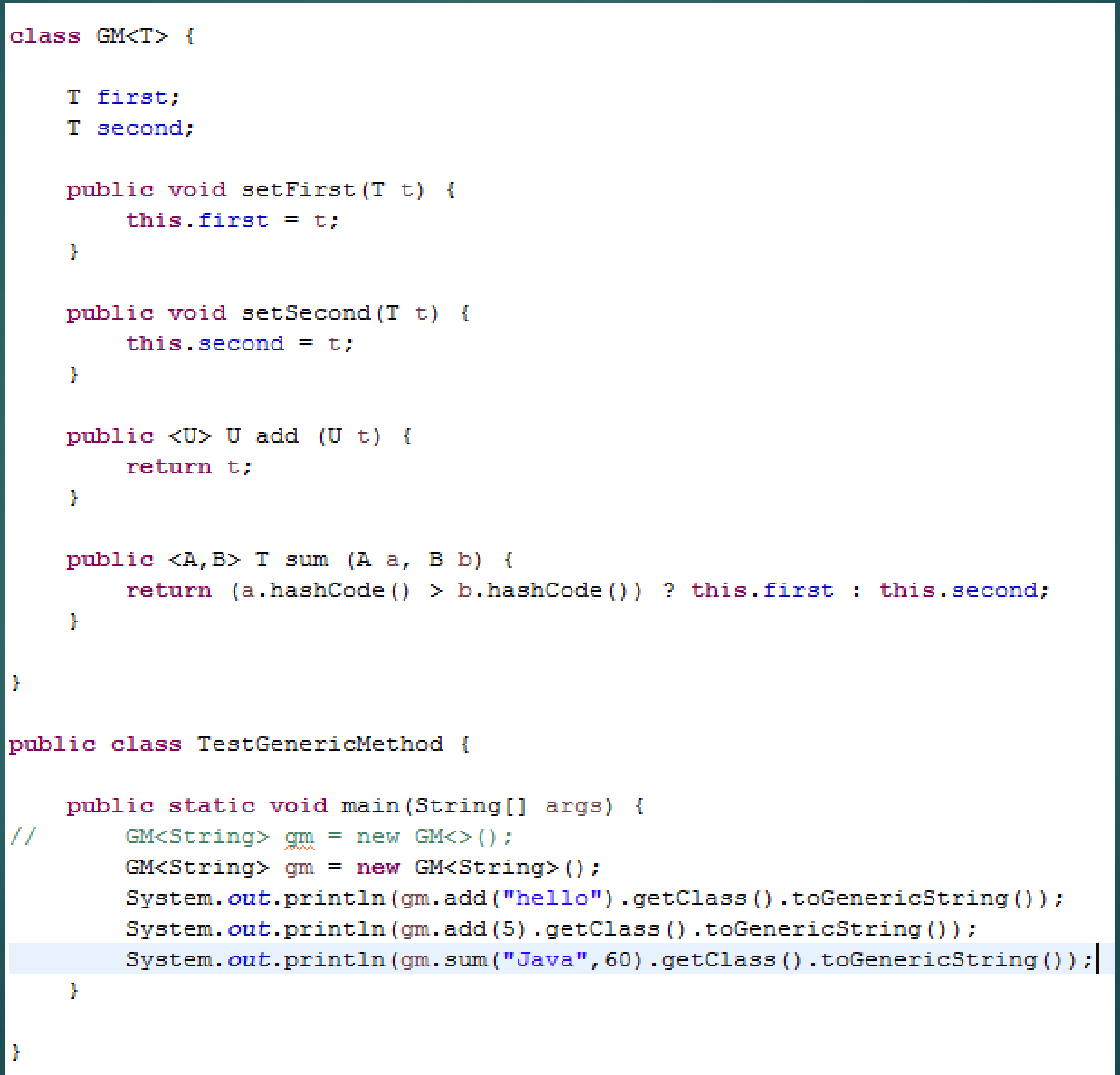
- The type parameter must be placed (in angular brackets) after all the modifiers, and before the returned type

```
public static <T> T genMethod(T[] a)
```

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets

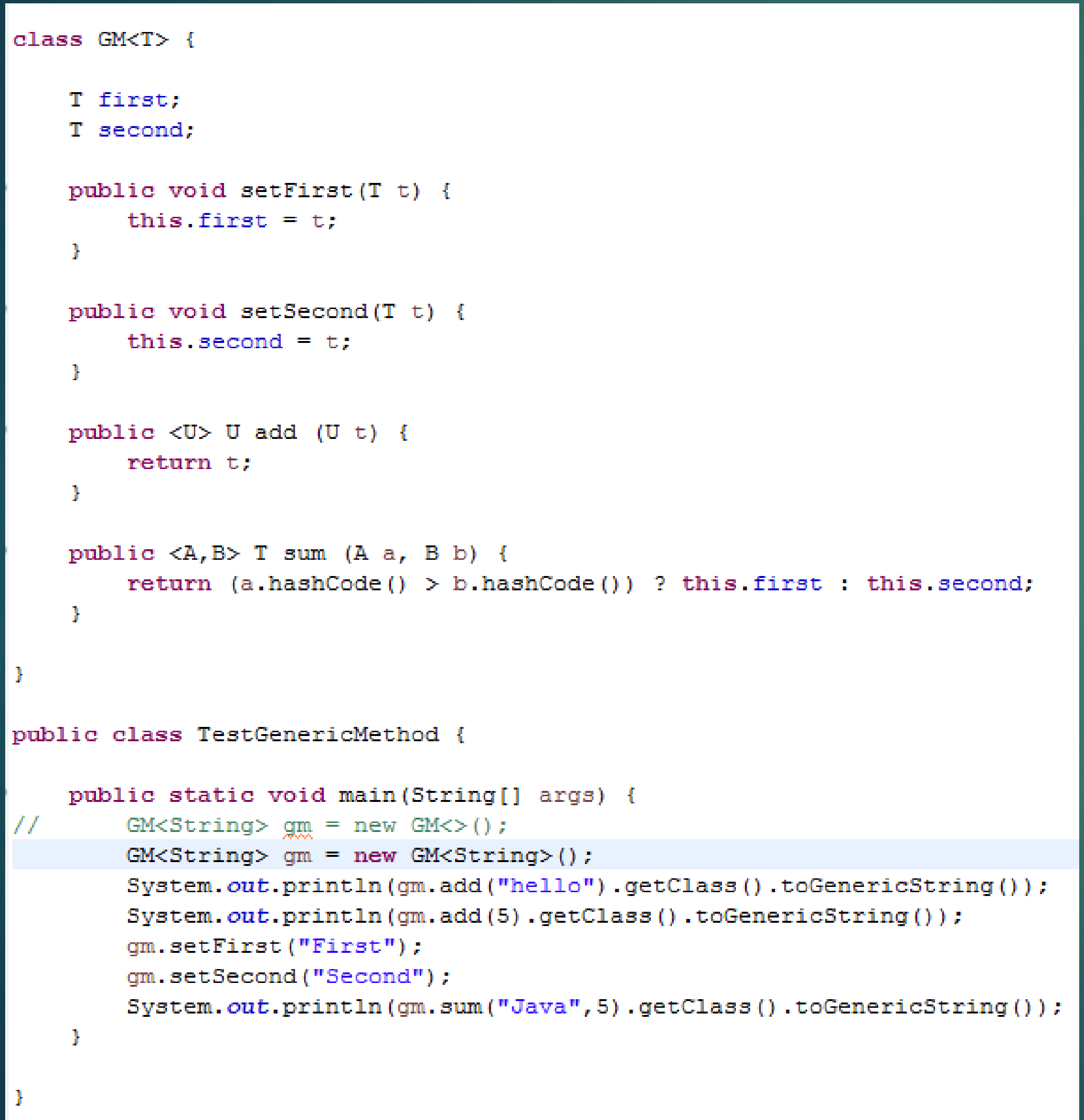
```
String s = NonG.<String>genMethod(c);
```

Generic Methods (cont’d)



```
class GM<T> {

    T first;
    T second;

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

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

    public <U> U add (U t) {
        return t;
    }

    public <A,B> T sum (A a, B b) {
        return (a.hashCode() > b.hashCode()) ? this.first : this.second;
    }

}

public class TestGenericMethod {

    public static void main(String[] args) {
//      GM<String> gm = new GM<>();
        GM<String> gm = new GM<String>();
        System.out.println(gm.add("hello").getClass().toGenericString());
        System.out.println(gm.add(5).getClass().toGenericString());
        System.out.println(gm.sum("Java",60).getClass().toGenericString());
    }

}
```

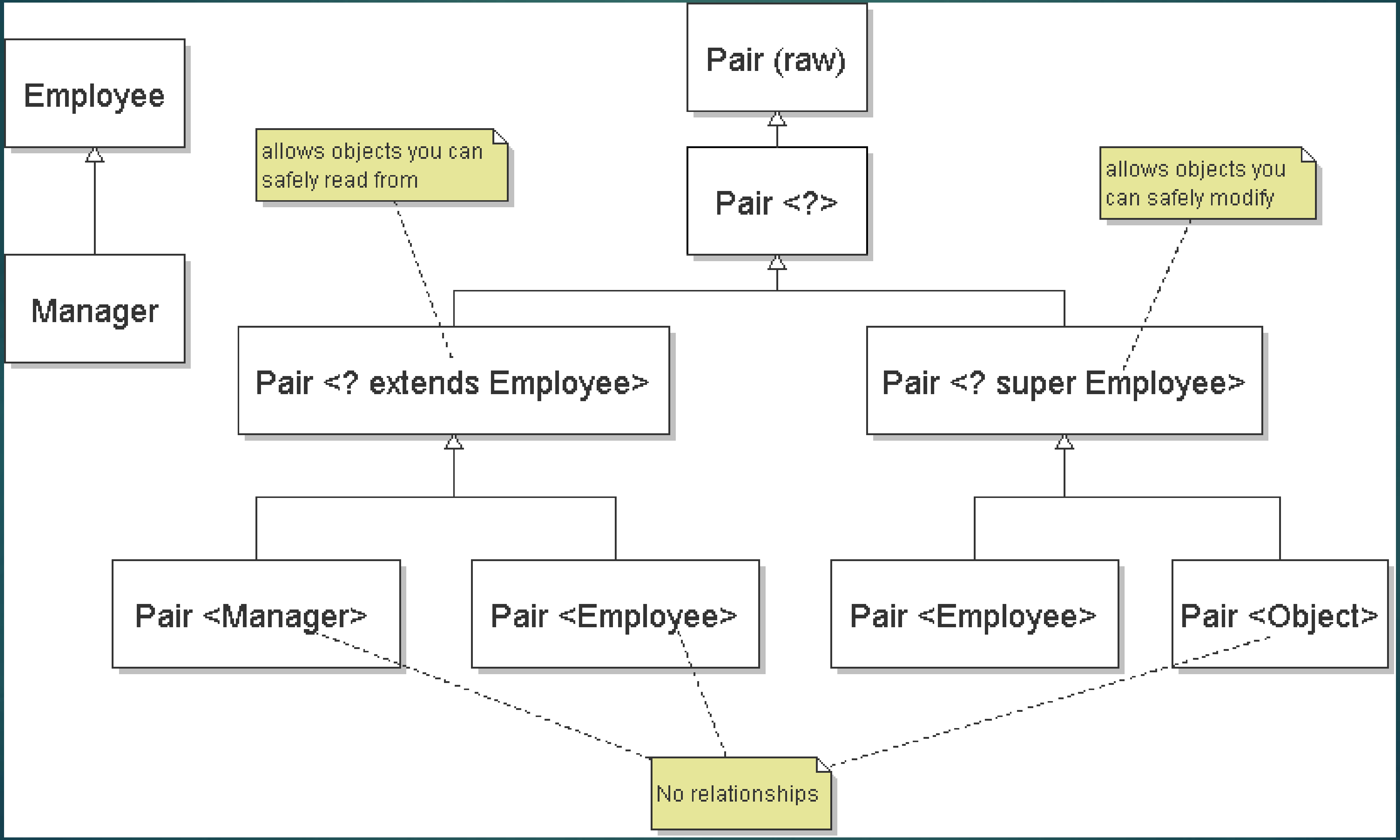
Generic Methods (cont'd)



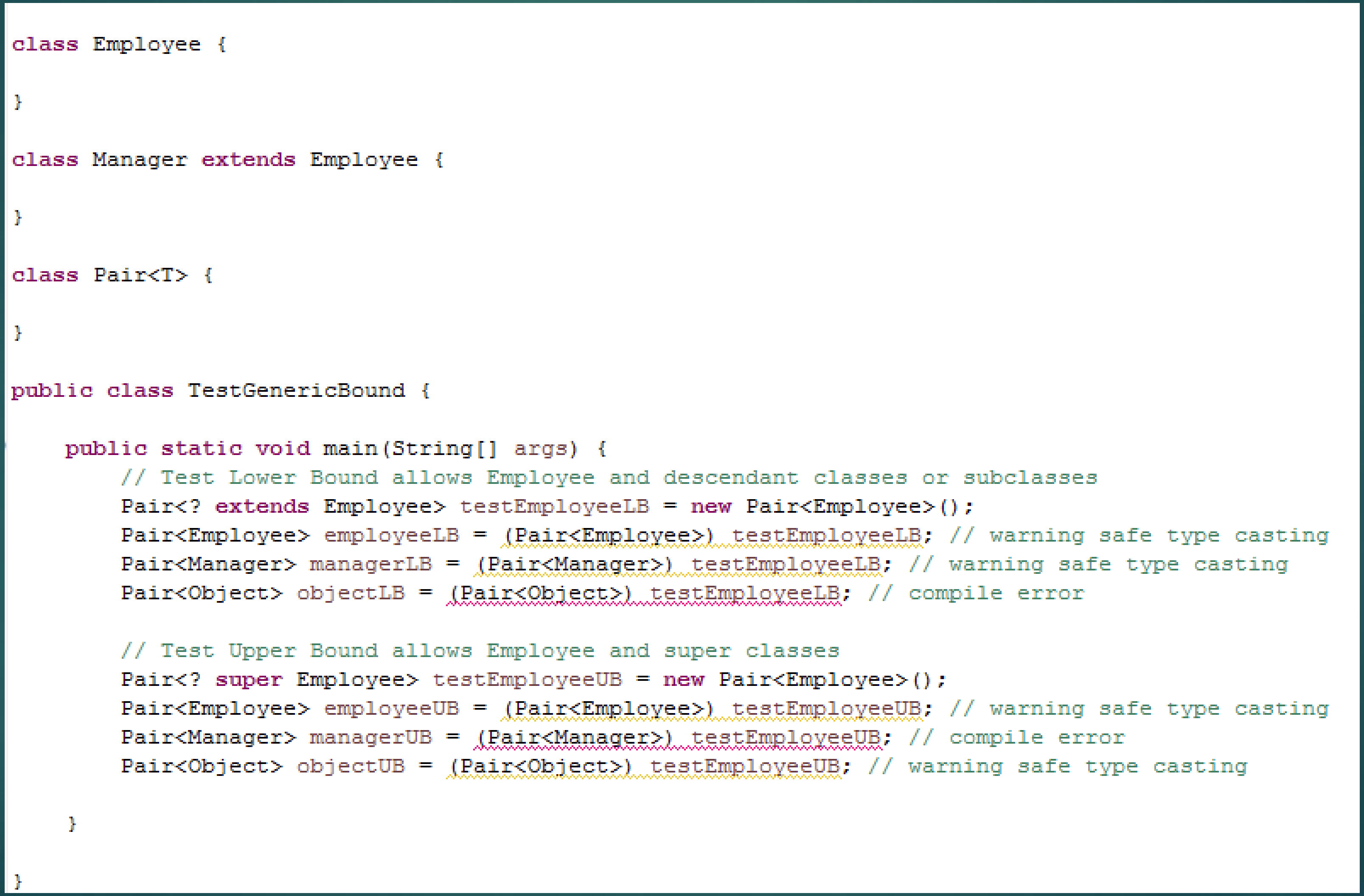
```
class GM<T> {

    T first;
    T second;

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

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

    public <U> U add (U t) {
        return t;
    }

    public <A,B> T sum (A a, B b) {
        return (a.hashCode() > b.hashCode()) ? this.first : this.second;
    }

}

public class TestGenericMethod {

    public static void main(String[] args) {
//      GM<String> gm = new GM<>();
        GM<String> gm = new GM<String>();
        System.out.println(gm.add("hello").getClass().toGenericString());
        System.out.println(gm.add(5).getClass().toGenericString());
        gm.setFirst("First");
        gm.setSecond("Second");
        System.out.println(gm.sum("Java",5).getClass().toGenericString());
    }

}
```

```
public final class java.lang.String
public final class java.lang.Integer
public final class java.lang.String
```

Bounded Type Parameter



Bounded Type Parameter Example

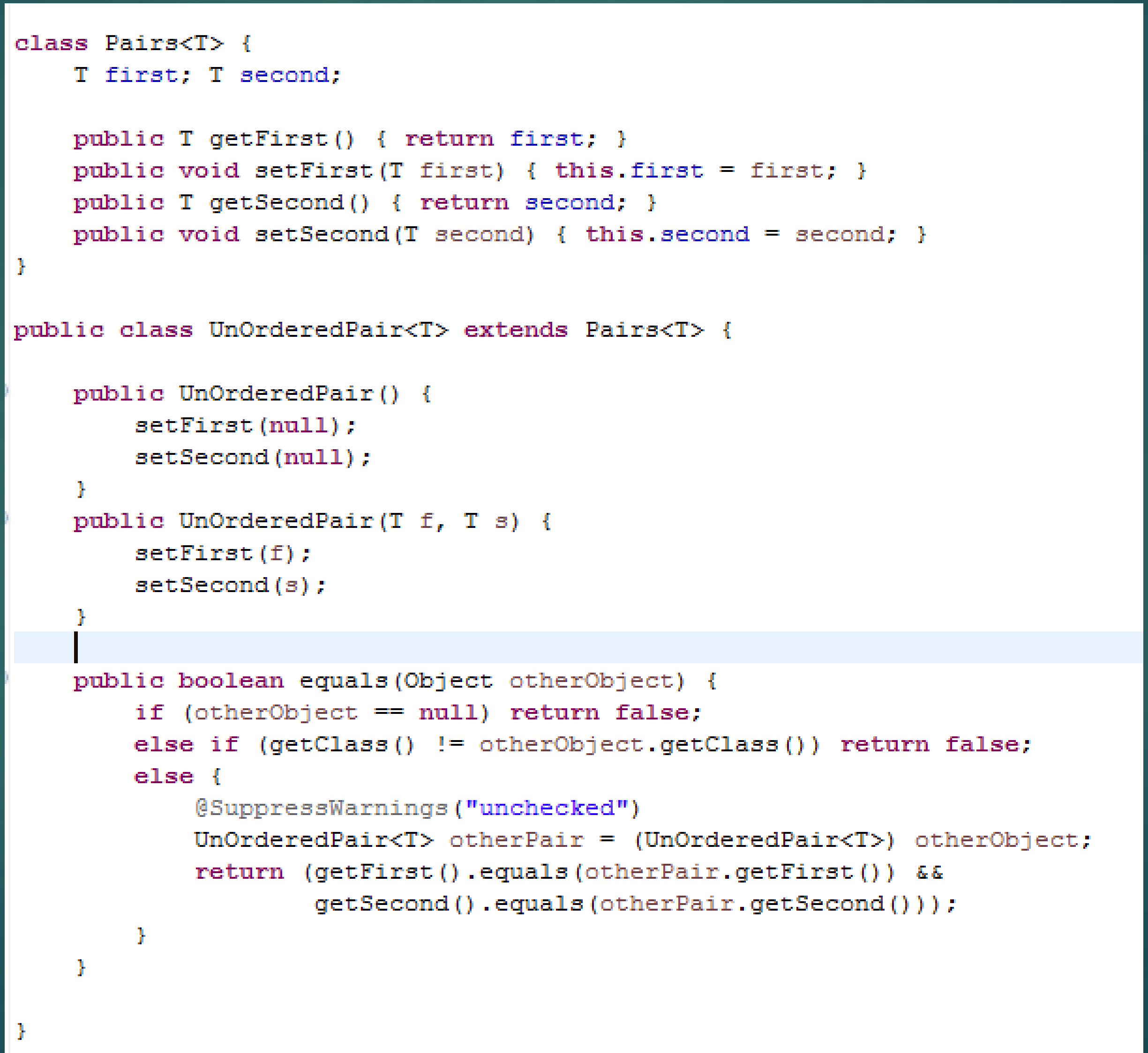


```
class Employee {

}

class Manager extends Employee {

}

class Pair<T> {

}

public class TestGenericBound {

    public static void main(String[] args) {
        // Test Lower Bound allows Employee and descendant classes or subclasses
        Pair<? extends Employee> testEmployeeLB = new Pair<Employee>();
        Pair<Employee> employeeLB = (Pair<Employee>) testEmployeeLB; // warning safe type casting
        Pair<Manager> managerLB = (Pair<Manager>) testEmployeeLB; // warning safe type casting
        Pair<Object> objectLB = (Pair<Object>) testEmployeeLB; // compile error

        // Test Upper Bound allows Employee and super classes
        Pair<? super Employee> testEmployeeUB = new Pair<Employee>();
        Pair<Employee> employeeUB = (Pair<Employee>) testEmployeeUB; // warning safe type casting
        Pair<Manager> managerUB = (Pair<Manager>) testEmployeeUB; // compile error
        Pair<Object> objectUB = (Pair<Object>) testEmployeeUB; // warning safe type casting

    }

}
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
- Given two classes: **`A`** and **`B`**, and given **G**: a generic class, there is no relationship between **`G<A>`** and **`G<B>`**
  - This is true regardless of the relationship between class **`A`** and **`B`**, e.g., if class **`B`** is a subclass of class **`A`**

Inheritance with Generic Classes (Cont'd)

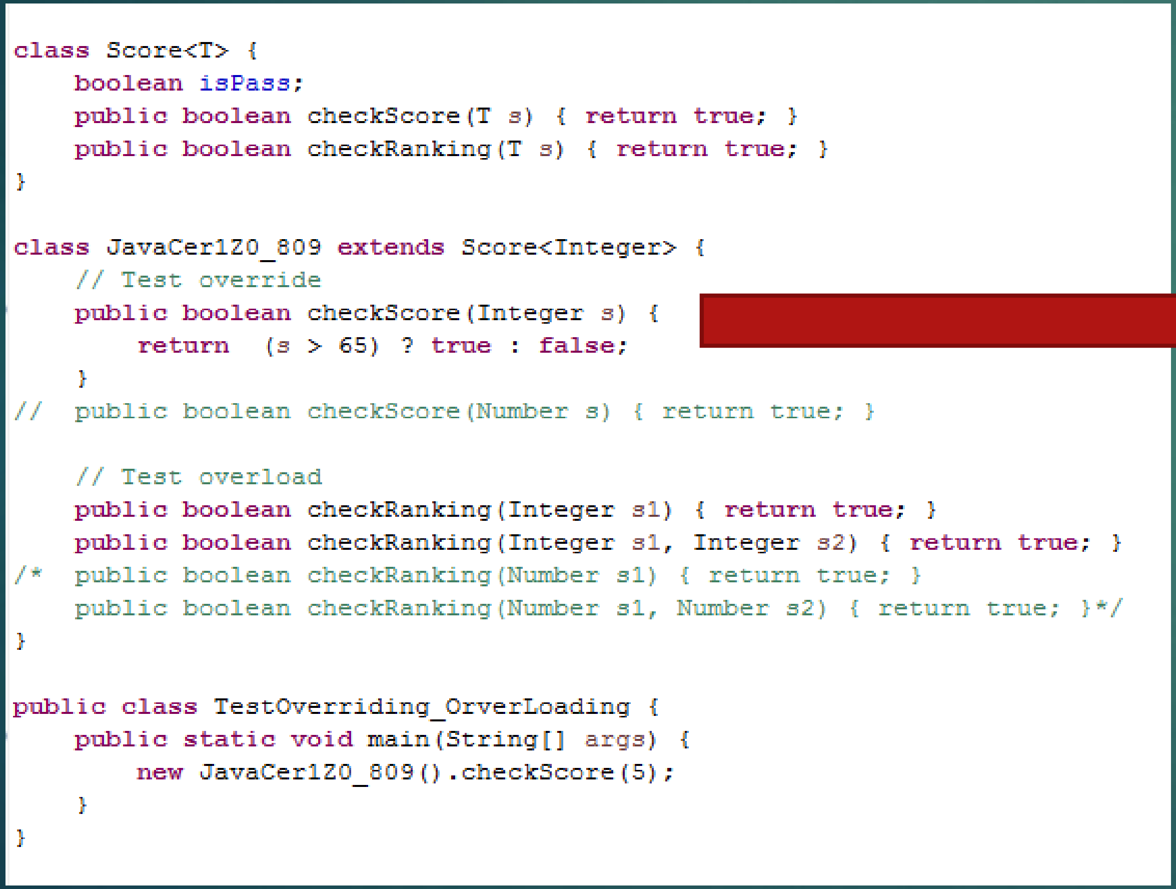


```
class Pairs<T> {
    T first; T second;

    public T getFirst() { return first; }
    public void setFirst(T first) { this.first = first; }
    public T getSecond() { return second; }
    public void setSecond(T second) { this.second = second; }
}

public class UnOrderedPair<T> extends Pairs<T> {

    public UnOrderedPair() {
        setFirst(null);
        setSecond(null);
    }
    public UnOrderedPair(T f, T s) {
        setFirst(f);
        setSecond(s);
    }

    public boolean equals(Object otherObject) {
        if (otherObject == null) return false;
        else if (getClass() != otherObject.getClass()) return false;
        else {
            @SuppressWarnings("unchecked")
            UnOrderedPair<T> otherPair = (UnOrderedPair<T>) otherObject;
            return (getFirst().equals(otherPair.getFirst()) &&
                    getSecond().equals(otherPair.getSecond()));
        }
    }

}
```

Overriding of methods of generic type

- consider a generic class with a non-final method:
- to override such type-erased methods, the compiler must generate extra *bridge methods*:

Overriding of methods of generic type example



```
class Score<T> {
    boolean isPass;
    public boolean checkScore(T s) { return true; }
    public boolean checkRanking(T s) { return true; }
}

class JavaCer1Z0_809 extends Score<Integer> {
    // Test override
    public boolean checkScore(Integer s) {
        return  (s > 65) ? true : false;
    }
//  public boolean checkScore(Number s) { return true; }

    // Test overload
    public boolean checkRanking(Integer s1) { return true; }
    public boolean checkRanking(Integer s1, Integer s2) { return true; }
/*  public boolean checkRanking(Number s1) { return true; }
    public boolean checkRanking(Number s1, Number s2) { return true; }*/
}

public class TestOverriding_OrverLoading {
    public static void main(String[] args) {
        new JavaCer1Z0_809().checkScore(5);
    }
}
```

```
public boolean checkScore(Object s) {

}
```

Overloading where type erasure will leave the parameters with the same type is not allowed:

```
class Test {
    // List<String> and List<Integer>
    // will be converted to List at runtime
    public void method(List<String> list) { }
    public void method(List<Integer> list) { }
}
```

Restriction and Limitations

A Generic Constructor Name Has No Type Parameter

- Although the class name in a parameterized class definition has a type parameter attached, the type parameter is not used in the heading of the constructor definition

```
public Pair<T>() // illegal
```

- A constructor can use the type parameter as the type for a parameter of the constructor, but in this case, the angular brackets are not used

```
public Pair(T first, T second)
```

- However, when a generic class is instantiated, the angular brackets are used

```
Pair<String> pair =
    new Pair<String>("Happy", "Day");
```

A Type Parameter Cannot Be Used Everywhere a Type Name Can Be Used

- In particular, the type parameter cannot be used in simple expressions using new to create a new object
  - For instance, the type parameter cannot be used as a constructor name or like a constructor:

```
T object = new T();
T[] a = new T[10];
```

An Instantiation of a Generic Class Cannot be an Array Base Type

- Arrays such as the following are illegal:

```
Pair<String>[] a =
   new Pair<String>[10]; // compile error
```

  - Although this is a reasonable thing to want to do, it is not allowed given the way that Java implements generic classes

Static fields and static method with type parameters are not allowed

- static fields and static methods with type parameters are not allowed

```
class Singleton <T> {
      private static T singleOne;        // ERROR
```

- since after type erasure, *one* class and *one* shared static field for all instantiations and their objects

Wildcard type (?) cannot be used as a declared type of any variables

- Pair<String> pair = new Pair<?>(); // Compile error

Generic types list

- These types can be made generic:
  - Classes
  - Interfaces
  - Inner classes, etc.
- These java types may not be generic:
  - Anonymous inner classes
  - Exceptions
  - Enums

A Generic Class Cannot Be an Exception Class

- It is not permitted to create a generic class with **Exception**, **Error**, **Throwable**, or any descendent class of **Throwable**
  - A generic class cannot be created whose objects are throwable

    public class GEx<T> extends Exception // error
  - The above example will generate a compiler error message
  - However, you can use a type parameter in a throws clause:

    -
      ```
      class Test<T extends Exception> {
          public void method() throws T { } // OK
      }
      ```

Conclusion

- Generics are a mechanism for type checking at compile-time.
- The process of replacing all references to generic types at runtime with an Object type is called type erasure.
- A generic class used without a generic type argument (like List list = null;) aka a raw type.
- The diamond operator (<>) can be used to simplify the use of generics when the type can be inferred by the compiler.
- It's possible to define a generic class or interface by declaring a type parameter next to the class or interface name.
- We can also declare type parameters in any method, specifying the type before the method return type (in contrast to classes, which declare it after the class name).
- The unbounded wildcard type (<?>) means that the type of the list is unknown so that it can match ANY type. This also means that for example, List<?> is a supertype of any List type (like List<Integer> or List<Float>).
- The upper-bounded wildcard (? extends T) means that you can assign either T or a subclass of T.
- The lower-bounded wildcard (? super T) means that you can assign either T or a superclass of T.