合肥學院
HEFEI UNIVERSITY

OOP with Java
21. Generics

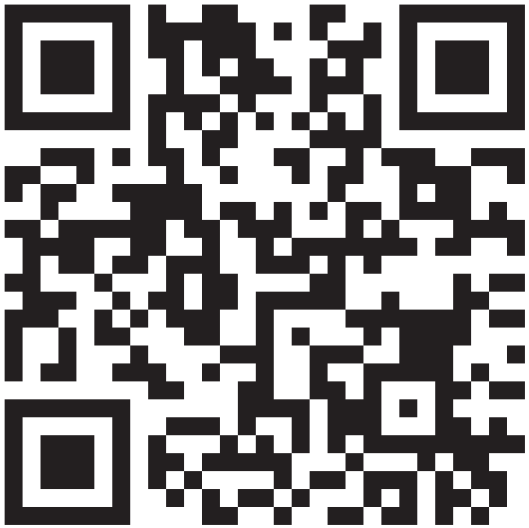
Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline



website

This is going to be a tough lesson.

Please listen carefully and ask questions whenever something is unclear.

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places, like the `Entry` class in the `Map` example in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places, like the `Entry` class in the `Map` example in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - Sometimes, you may want to use your class to store:
    - `String` - `String` associations

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places, like the `Entry` class in the `Map` example in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - Sometimes, you may want to use your class to store:
    - `String` - `String` associations
    - `Integer` - `String` associations

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places, like the `Entry` class in the `Map` example in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - Sometimes, you may want to use your class to store:
    - `String` - `String` associations
    - `Integer` - `String` associations
    - other stuff

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places, like the `Entry` class in the `Map` example in Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*
  - Sometimes, you may want to use your class to store:
    - `String` - `String` associations
    - `Integer` - `String` associations
    - other stuff
  - but then you will always need to use type casts (see Lesson 20) when reading the key/value instance variables

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places
  - but then you will always need to use type casts (see Lesson 20) when reading the key/value instance variables
- This creates several problems

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places
  - but then you will always need to use type casts (see Lesson 20) when reading the key/value instance variables
- This creates several problems:
  - you may sometimes do a wrong type cast and the compiler cannot check whether you use the right types

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places
  - but then you will always need to use type casts (see Lesson 20) when reading the key/value instance variables
- This creates several problems:
  - you may sometimes do a wrong type cast and the compiler cannot check whether you use the right types
  - this also means more code, more code = harder to read and maintain

Introduction

- Imagine you want to create a class for holding a pair of objects, say a key-value association
  - this seems to be a rather general utility class
  - ideally, you want this class to be useful for any kind of key/value object
  - you want that the algorithms you implement using this class can be applied to any kind of key/value objects
  - so you would implement it using `Object` key/value instance variables
  - then you can use your class in many different places
  - but then you will always need to use type casts (see Lesson 20) when reading the key/value instance variables
- This creates several problems:
  - you may sometimes do a wrong type cast and the compiler cannot check whether you use the right types
  - this also means more code, more code = harder to read and maintain
- Let's look at an example

A class for holding a pair of objects

Listing: A class for holding a pair of objects

```
package cn.edu.hfuu.iao.collections;

/** a non-generic, Object-based key-value pair */
public class Pair {
  /** the key object */
  public final Object key;
  /** the value object */
  private Object value;

  /** create */
  public Pair(final Object _key, final Object _value) {
    this.key = _key;
    this.value = _value;
  }

  /** set the value */
  public void setValue(final Object _value) {
    this.value = _value;
  }

  /** get the value */
  public Object getValue() {
    return this.value;
  }
}
```

A use case for the class for holding a pair of objects

Listing: A use case for the class for holding a pair of objects

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.Pair;

/** a class where we use the Object-based Pair class */
public class PairTest {
  /** The main routine
   *  @param args we ignore this parameter */
  public static void main(String[] args) {
    Pair stringPair = new Pair("Hello", "World!"); //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(stringPair.key); // Hello
    System.out.println(stringPair.getValue()); // World!

    // Integer is a java utility class, its instance can hold
    Pair stringIntegerPair = new Pair("int", new Integer(3));  //$NON-NLS-1$
    System.out.println(stringIntegerPair.key); // "int"
    System.out.println(stringIntegerPair.getValue()); // 3

    // String keyString = stringPair.key; // not allowed, key could be any Object
    String keyString = (String) (stringPair.key); // we need explicit casting
    System.out.println(keyString); // Hello

    // String valueString = stringPair.getValue(); // not allowed, value could be any object
    String valueString = (String) (stringPair.getValue()); // we need explicit casting
    System.out.println(valueString); // World!

    stringIntegerPair = stringPair; // this is allowed
    System.out.println(stringIntegerPair.key);  //  Hello
    System.out.println(stringIntegerPair.getValue()); // World!
  }
}
```

The Problem

- We know that `stringPair` contains two Strings, and by its name, we clearly intent it to only hold two strings

The Problem

- We know that `stringPair` contains two Strings, and by its name, we clearly intent it to only hold two strings
- The `stringIntegerPair` holds a String and an Integer, and it is intended for this

The Problem

- We know that `stringPair` contains two Strings, and by its name, we clearly intent it to only hold two strings
- The `stringIntegerPair` holds a String and an Integer, and it is intended for this
- We want to use class `Pair` for both of these objects, because, well, it sort of fits

The Problem

- We know that `stringPair` contains two Strings, and by its name, we clearly intent it to only hold two strings
- The `stringIntegerPair` holds a String and an Integer, and it is intended for this
- We want to use class `Pair` for both of these objects, because, well, it sort of fits
- But this provides no type safety, we would need to use `instanceof` and explicit type casts all the time

The Problem

- We know that `stringPair` contains two Strings, and by its name, we clearly intent it to only hold two strings
- The `stringIntegerPair` holds a String and an Integer, and it is intended for this
- We want to use class `Pair` for both of these objects, because, well, it sort of fits
- But this provides no type safety, we would need to use `instanceof` and explicit type casts all the time
- And we cannot really control the types of the stuff actually stored in the pair if it comes from elsewhere

Generics

- Classes can have (arbitrarily many) type parameters

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like “real types” inside the class

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified
- We can create a generic class for pairs

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified
- We can create a generic class for pairs
- The class could have two type parameters `K` and `V`, one (`K`) for the key type, one (`V`) for the value type

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified
- We can create a generic class for pairs
- The class could have two type parameters `K` and `V`, one (`K`) for the key type, one (`V`) for the value type
- When implementing the class, we can then use `K` and `V` as if they were normal class types

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified
- We can create a generic class for pairs
- The class could have two type parameters `K` and `V`, one (`K`) for the key type, one (`V`) for the value type
- When implementing the class, we can then use `K` and `V` as if they were normal class types
- When instantiating the generic class, we need to provide concrete types as replacement for the `K` and `V`, say `String` and `Integer`.

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified
- We can create a generic class for pairs
- The class could have two type parameters `K` and `V`, one (`K`) for the key type, one (`V`) for the value type
- When implementing the class, we can then use `K` and `V` as if they were normal class types
- When instantiating the generic class, we need to provide concrete types as replacement for the `K` and `V`, say `String` and `Integer`.
- The instances are then only assignment compatible if they have the same type replacements

Generics

- Classes can have (arbitrarily many) type parameters
- The type parameters can (almost) be used like "real types" inside the class
- When instantiating the class, the actual types of the type parameters must be specified
- We can create a generic class for pairs
- The class could have two type parameters `K` and `V`, one (`K`) for the key type, one (`V`) for the value type
- When implementing the class, we can then use `K` and `V` as if they were normal class types
- When instantiating the generic class, we need to provide concrete types as replacement for the `K` and `V`, say `String` and `Integer`.
- The instances are then only assignment compatible if they have the same type replacements
- Generic parameters must be classes, they can never be primitive types! (because of erasure, see later)

A generic class for holding a pair of objects

Listing: A generic class for holding a pair of objects

```java
package cn.edu.hfuu.iao.collections;

/**
 * a generic key-value pair where we can specify the types
 * @param <K>   the generic key type
 * @param <V>   the generic value type */
public class GenericPair<K, V> {
  /** the key object */
  public final K key;
  /** the value object */
  private V value;

  /** create */
  public GenericPair(final K _key, final V _value) {
    this.key = _key;
    this.value = _value;
  }

  /** set the value: must be of type V */
  public void setValue(final V _value) {
    this.value = _value;
  }

  /** get the value */
  public V getValue() {
    return this.value;
  }
}
```

- The generic type parameters `K` and `V` are in some sense parameters of a class

Javadoc

- The generic type parameters `K` and `V` are in some sense parameters of a class
- We can describe their meaning in Javadoc comments in the form of `@param <K> meaning of K` and `@param <V> meaning of V`

A use case for the generic class for holding a pair of objects

Listing: A use case for the generic class for holding a pair of objects

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.GenericPair;

/** a class where we use the GenericPair class */
public class GenericPairTest {
  /** The main routine
   *  @param args we ignore this parameter */
  public static void main(String[] args) {
    GenericPair<String,String> stringPair = //
             new GenericPair<String,String>("Hello", "World!"); //$NON-NLS-1$ //$NON-NLS-2$

    System.out.println(stringPair.key); // Hello
    System.out.println(stringPair.getValue()); // World!

    GenericPair<String,Integer> stringIntegerPair =   // we can use <> (instead of <String,Integer>)
            new GenericPair<>("int", new Integer(3)); // if the generic parameters are clear //$NON-NLS-1$
    System.out.println(stringIntegerPair.key);        // "int"
    System.out.println(stringIntegerPair.getValue()); // 3

    String keyString = stringPair.key; // we do no longer need need explicit casting
    System.out.print(keyString);

    String valueString = stringPair.getValue(); // we do no longer need explicit casting
    System.out.print(valueString);

    // integerPair = stringPair; // this is now forbidden
  }
}
```

A use case for a generic object holding a generic object

Listing: A use case for a generic object holding a generic object

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.GenericPair;

/** a class where we use the GenericPair class inside a generic pair */
public class GenericPairParameterizedParametersTest {

  /** The main routine
   *  @param args we ignore this parameter */
  public static void main(String[] args) {
    GenericPair<String, Integer> stringIntegerPair;
    GenericPair<String, GenericPair<String, Integer>> wrappedPair;

    stringIntegerPair =                              // we can use <> (instead of <String, GenericPair<String,
        Integer>>)
        new GenericPair<>("int", new Integer(3)); // since the generic parameters are clear //$NON-NLS-1$

    wrappedPair = new GenericPair<>("Hello", stringIntegerPair); //$NON-NLS-1$
    System.out.println(wrappedPair.key);                      // Hello
    System.out.println(wrappedPair.getValue().key);           // int
    System.out.println(wrappedPair.getValue().getValue());    // 3

    Integer integer = wrappedPair.getValue().getValue();      // no type case to Integer necessary
    System.out.println(integer);                              // 3

    wrappedPair.getValue().setValue(new Integer(6));          // this only is allowed with Integer
    System.out.println(wrappedPair.getValue().getValue());    // 6

    wrappedPair.setValue(new GenericPair<>("newInt", new Integer(7)));  //$NON-NLS-1$
    System.out.println(wrappedPair.key);                      // Hello
    System.out.println(wrappedPair.getValue().key);           // newInt
    System.out.println(wrappedPair.getValue().getValue());    // 7
  }
}
```

Advantages

- Generics have serval advantages

Advantages

- Generics have serval advantages:
  - stronger type checks at compile time

Advantages

- Generics have serval advantages:
  - stronger type checks at compile time
  - reduce the number of type casts / need for `instanceof`

Advantages

- Generics have serval advantages:
  - stronger type checks at compile time
  - reduce the number of type casts / need for `instanceof`
  - allow us to implement generic algorithms and data structures without sacrificing type safety

A more elaborate example

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we did an example on with a `Map` where we can store key-value relationships

A more elaborate example

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we did an example on with a `Map` where we can store key-value relationships
- Let us take a look at it again (this time using our `Pair` classes internally)

A more elaborate example

- In Lesson 18: *Visibility, Encapsulation,* `final`*, and Inner Classes*, we did an example on with a `Map` where we can store key-value relationships
- Let us take a look at it again (this time using our `Pair` classes internally)
- And then make it generic

A non-generic/`Object`-based `Map` Class

Listing: Class representing a non-generic/`Object`-based `Map`

```
package cn.edu.hfuu.iao.collections;

/** a non-generic, Object-based map which stores key-value relationships */
public final class Map {

  /** the entry list: see the private class Pair below */
  private Pair[] entries;

  /** create a map */
  public Map() {                  // create the map
    this.entries = new Pair[32]; // with space for some entries
  }

  /** store that key should now be related to value */
  public final void put(final Object key, final Object value) {
    for (int index = 0; index < this.entries.length; index++) {      // first check all stored keys
      if (this.entries[index] == null) {                             // if we get here, we have reached the end of the map
        this.entries[index] = new Pair(key, value);                  // since we did not find key, just put a new entry
        return; // and we can exit
      }
      if (this.entries[index].key == key) {                          // check if there already is an entry for the specified key
        this.entries[index].setValue(value);                         // if so, we need to change its associated value
        return;                                                      // and can exit
      }
    } // if we get to after the loop, this means that the entry list is full, but does not contain key
    Pair[] newEntries = new Pair[this.entries.length * 2];           // so we need to allocate a larger entry list
    for(int i = this.entries.length; (--i) >= 0; ) { newEntries[i] = this.entries[i]; } // copy all existing entries
    newEntries[this.entries.length] = new Pair(key, value);          // and at the end of this list, we put the new entry
    this.entries = newEntries;                                       // and store the new entry list
  }

  /** get the value associated with the given key, or null if nothing is stored under the key */
  public final Object get(final Object key) {
    for (Pair entry : this.entries) {                                 // simply iterate over all entries
      if (entry == null) { return null; }                             // we reached the end and found nothing, return null
      if (entry.key == key) { return entry.getValue(); }              // we found an entry with the right key, return the value
    }
    return null;                                                      // we found nothing, let's return null
  }

  /** transform to string */
  public final String toString() {
    String string = ""; //$NON-NLS-1$
    for (Pair entry : this.entries) {                               // fast iteration over all entries
      if (entry == null) { return string; }                         // end of list reached
      if (string != "") { string += ", "; } //$NON-NLS-1$ //$NON-NLS-2$
      string += entry.key + "=" + entry.getValue();                 //add key-value relationship //$NON-NLS-1$
    }
    return string; // return string
  }
}
```

Example for Using our non-generic/`Object`-based `Map`

Listing: Class using our non-generic/`Object`-based `Map` class

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.Map;

/** a class where we use our map */
public class MapTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    Map map = new Map();

    map.put("Hello", "World!");              //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map);                 // Hello=World!

    map.put("Country", "China");             //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map);                 // Hello=World!, Country=China
    System.out.println(map.get("Country")); // China //$NON-NLS-1$
    System.out.println(map.get("Hello"));    // World //$NON-NLS-1$
    System.out.println(map.get("World!"));   // null, World! is not a key //$NON-NLS-1$

    map.put("Computer Science", "Fun");      //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map);                 // Hello=World!, Country=China, Computer Science=Fun

    // String str = map.get("Hello");        // this does not work
    String str = (String)(map.get("Hello"));// we need to cast //$NON-NLS-1$
    System.out.println(str);                 // World!

    Object obj1 = str;
    Object obj2 = "You";                     //$NON-NLS-1$
    map.put(obj1, obj2);                     // this is allowed
    System.out.println(map.get(obj1));       // You
  }
}
```

Listing: A generic class representing a Map

```
package cn.edu.hfuu.iao.collections;

/** a map which stores key-value relationships of keys and values of specific types
 * @param <K> the generic key type
 * @param <V> the generic value type*/
public final class GenericMap<K, V> {

  /** the entry list: see the private class GenericPair below */
  private GenericPair<K,V>[] entries;

  /** create a map */
  @SuppressWarnings("unchecked")
  public GenericMap() {              // create the map
    this.entries = new GenericPair[32]; // with space for some entries
  }

  /** store that key should now be related to value */
  @SuppressWarnings("unchecked")
  public final void put(final K key, final V value) {               // we here use the generic types K and V
    for (int index = 0; index < this.entries.length; index++) {     // first check all stored keys
      if (this.entries[index] == null) {                            // if we get here, we have reached the end of the map
        this.entries[index] = new GenericPair<>(key, value);              // since we did not find key, just put a new entry
        return; // and we can exit
      }
      if (this.entries[index].key == key) {                         // check if there already is an entry for the specified key
        this.entries[index].setValue(value);                          // if so, we need to change its associated value
        return;                                                     // and can exit
      }
    } // if we get to after the loop, this means that the entry list is full, but does not contain key
    GenericPair<K, V>[] newEntries = new GenericPair[this.entries.length * 2];  // so we need to allocate a larger entry list
    for(int i = this.entries.length; (--i) >= 0; ) { newEntries[i] = this.entries[i]; } // copy all existing entries
    newEntries[this.entries.length] = new GenericPair<>(key, value);      // and at the end of this list, we put the new entry
    this.entries = newEntries;                                      // and store the new entry list
  }

  /** get the value associated with the given key, or null if nothing is stored under the key */
  public final V get(final K key) {
    for (GenericPair<K,V> entry : this.entries) {                    // simply iterate over all entries
      if (entry == null) { return null; }                            // we reached the end and found nothing, return null
      if (entry.key == key) { return entry.getValue(); }             // we found an entry with the right key, return the value
    }
    return null;                                                     // we found nothing, let's return null
  }

  /** transform to string */
  public final String toString() {
    String string = ""; //$NON-NLS-1$
    for (GenericPair<K, V> entry : this.entries) {                   // fast iteration over all entries
      if (entry == null) { return string; }                          // end of list reached
      if (string != "") { string += ",␣"; } //$NON-NLS-1$ //$NON-NLS-2$
      string += entry.key + "=" + entry.getValue();                 //add key-value relationship //$NON-NLS-1$
    }
    return string; // return string
  }
}
```

Example for Using our Generic `Map`

Listing: Class using our generic `Map` class

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.GenericMap;

/** a class where we use our generic map */
public class GenericMapTest {
  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    GenericMap<String,String> map = new GenericMap<>();

    map.put("Hello", "World!");              //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map);                 // Hello=World!

    map.put("Country", "China");             //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map);                 // Hello=World!, Country=China
    System.out.println(map.get("Country")); // China //$NON-NLS-1$
    System.out.println(map.get("Hello"));    // World //$NON-NLS-1$
    System.out.println(map.get("World!"));   // null, World! is not a key //$NON-NLS-1$

    map.put("Computer Science", "Fun");      //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(map);                 // Hello=World!, Country=China, Computer Science=Fun

    String str = map.get("Hello");           // this does now work, we do no longer need to cast //$NON-NLS-1$
    System.out.println(str);                 // World!

    Object obj1 = str;
    Object obj2 = "You";                     //$NON-NLS-1$
    // map.put(obj1, obj2);                  // this is no longer allowed, the parameters must be Strings
    map.put((String)obj1, (String)obj2);     // we now need to type cast
    // System.out.println(map.get(obj1));    // this is no longer allowed, keys need to be Strings
    System.out.println(map.get((String)obj1)); // You
  }
}
```

Methods with Generic Parameters

- Methods can have generic type parameters as well, like classes

Methods with Generic Parameters

- Methods can have generic type parameters as well, like classes
- These then need to be specified before the return type

Example for methods with generic parameters

Listing: Example for methods with generic parameters

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.GenericPair;

/** a class where we use the GenericPair class and implement a method with generic parameters */
public class GenericsAndStaticFunctions {

  /** a static function can have generic parameters as well */
  static <K, V> GenericPair<K, V> makePair(final K key, final V value) {
    return new GenericPair<>(key, value); // <> is used, since the generics are clear
  }

  /** a static function checking if two pairs have exactly the same parameters */
  static <K, V> boolean isSame(GenericPair<K, V> pair1, GenericPair<K, V> pair2) {
    return ((pair1.key == pair2.key) && (pair1.getValue() == pair2.getValue()));
  }

  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    GenericPair<String, String> stringPair = makePair("Hello", "World!"); //$NON-NLS-1$ //$NON-NLS-2$

    System.out.println(stringPair.key); // Hello
    System.out.println(stringPair.getValue()); // World!

    GenericPair<String, Integer> stringIntegerPair = makePair("int", new Integer(3)); //$NON-NLS-1$
    System.out.println(stringIntegerPair.key); // "int"
    System.out.println(stringIntegerPair.getValue()); // 3

    String keyString = stringPair.key; // we do no longer need need explicit casting
    System.out.println(keyString); // Hello

    String valueString = stringPair.getValue(); // we do no longer need explicit casting
    System.out.println(valueString); // World!

    // integerPair = stringPair; // this is now forbidden

    System.out.println(isSame(stringPair, makePair("Hello", "World!"))); // true //$NON-NLS-1$ //$NON-NLS-2$
    System.out.println(isSame(stringPair, makePair("Hello", "You")));    // false //$NON-NLS-1$ //$NON-NLS-2$

    // System.out.println(isSame(stringPair, stringIntegerPair));        // this is forbidden altogether
  }
}
```

Lower Bounds for Type Parameters

- We can define lower bounds for a type parameter

Lower Bounds for Type Parameters

- We can define lower bounds for a type parameter
- Normal type parameter: `class A<B> { ...`

Lower Bounds for Type Parameters

- We can define lower bounds for a type parameter
- Normal type parameter: `class A<B> { ...`
- Type parameter with lower bound `C`: `class A<B extends C> { ...`

Lower Bounds for Type Parameters

- We can define lower bounds for a type parameter
- Normal type parameter: `class A<B> { ...`
- Type parameter with lower bound `C`: `class A<B extends C> { ...`
- Meaning: class `A` can only be instantiate with a value for `B` which is either class `C` itself or another class `D` which is a direct or indirect subclass of `C`

A base class for our example

Listing: Class Printable

```
package cn.edu.hfuu.iao.bounds;

/** a base class for all printable objects */
public class Printable {

  /** print this object */
  public void print() {
    System.out.println("this␣is␣a␣printable␣object"); //$NON-NLS-1$
  }
}
```

The first subclass for our example

Listing: Class FunnyPrintable

```
package cn.edu.hfuu.iao.bounds;

/** a funny printable */
public class FunnyPrintable extends Printable {
  /** print this object */
  @Override
  public void print() {
    System.out.println("Whats␣the␣object-oriented␣way␣to␣become␣wealthy?␣Inheritance"); //$NON-NLS-1$
  }
}
```

The second subclass for our example

Listing: Class MathPrintable

```
package cn.edu.hfuu.iao.bounds;

/** a funny printable */
public class MathPrintable extends Printable {
  /** the integer */
  private final int number;

  public MathPrintable(int _number) { this.number = _number; }

  /** print this object */
  @Override
  public void print() {
    System.out.println(this.number);
  }
}
```

The generic container for two printables (with lower bound)

Listing: Class TwoPrintables

```
package cn.edu.hfuu.iao.bounds;

/** a class of two printables is a pair whose elements must be printable */
public class TwoPrintables<T extends Printable> extends Printable {
  /** the first printable */
  private final T a;
  /** the second printable */
  private final T b;

  /** create */
  public TwoPrintables(final T _a, final T _b) {
    this.a = _a; this.b = _b;
  }

  /** print this object */
  @Override
  public void print() {// since the lower bound for T is Printable, we
    this.a.print();     // can be sure that a and b have a method "print"
    this.b.print();     // and thus we can use it. Without lower bound,
  }                     // this would not have been possible, since class
}                       // Object does not have such a method
```

An example for instantiating with lower-bound generic types

Listing: Class TwoPrintablesTest

```
package cn.edu.hfuu.iao.bounds;

/** a class where we use the printables with lower type bounds */
public class TwoPrintablesTest {
  /** The main routine
   *  @param args we ignore this parameter */
  public static void main(String[] args) {
    FunnyPrintable funny = new FunnyPrintable();
    MathPrintable  math1 = new MathPrintable(1);
    MathPrintable  math2 = new MathPrintable(2);

    funny.print(); // Whats the object-oriented way to become wealthy? Inheritance
    math1.print(); // 1
    math2.print(); // 2

    TwoPrintables<Printable> two1 = new TwoPrintables<>(funny, math1);
    two1.print(); // Whats the object-oriented way to become wealthy? Inheritance\n1

    TwoPrintables<MathPrintable> two2 = new TwoPrintables<>(math1, math2);
    two2.print(); // 1 \n 2

    // TwoPrintables<MathPrintable> two2 = new TwoPrintables<>(math1, funny); // not allowed

    TwoPrintables<TwoPrintables<MathPrintable>> four = new TwoPrintables<>(
        new TwoPrintables<>(new MathPrintable(1), new MathPrintable(2)),
        new TwoPrintables<>(new MathPrintable(3), new MathPrintable(4)));
    four.print(); // 1\n\2\n\3\n\4
  }
}
```

Upper Bounds and Wildcards for Type Parameters

- We can define lower bounds for a type parameter in the form

```
class A<B extends C> { ...
```

Upper Bounds and Wildcards for Type Parameters

- We can define lower bounds for a type parameter in the form `class A<B extends C> { ...`
- It makes no real sense to define an upper bound `U` for a class type parameter, as this would mean that we can store anything `super U` in there, starting of `Object`

Upper Bounds and Wildcards for Type Parameters

- We can define lower bounds for a type parameter in the form `class A<B extends C> { ...`
- It makes no real sense to define an upper bound `U` for a class type parameter, as this would mean that we can store anything `super U` in there, starting of `Object`
- But sometimes, in type-parameterized methods, we may need do deal with this range

Upper Bounds and Wildcards for Type Parameters

- We can define lower bounds for a type parameter in the form `class A<B extends C> { ...`
- It makes no real sense to define an upper bound `U` for a class type parameter, as this would mean that we can store anything `super U` in there, starting of `Object`
- But sometimes, in type-parameterized methods, we may need do deal with this range
- Also, sometimes, we may not really care about the actual type of a parameter, as long as it obeys a certain lower bound

Upper Bounds and Wildcards for Type Parameters

- We can define lower bounds for a type parameter in the form `class A<B extends C> { ...`
- It makes no real sense to define an upper bound `U` for a class type parameter, as this would mean that we can store anything `super U` in there, starting of `Object`
- But sometimes, in type-parameterized methods, we may need do deal with this range
- Also, sometimes, we may not really care about the actual type of a parameter, as long as it obeys a certain lower bound
- In both cases, we can use the wildcard `?` (which is not to be confused with the ternary operator from Lesson 5: *Operators Expressions*)

An example for wildcards for generic types

Listing: An example for wildcards for generic types

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.GenericPair;

/** a class where we use wildcards on the GenericPair class and implement a method with generic parameters */
public class GenericsAndStaticFunctionsWildcards {

  /** a static function creating a copy of a pair with possible super key and value types  */
  static <K, V> GenericPair<K, V> copyPair(final GenericPair<? extends K, ? extends V> pair) {
    return new GenericPair<>(pair.key, pair.getValue()); // <> is used, since the generics are clear
  }

  /** a static function checking if two pairs have exactly the same parameters */
  static <K, V> boolean isSame(GenericPair<K, V> pair1, GenericPair<? super K, ? super V> pair2) {
    return ((pair1.key == pair2.key) && (pair1.getValue() == pair2.getValue()));
  }

  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    GenericPair<String, String> stringPair = new GenericPair<>("Hello", "World!"); //$NON-NLS-1$ //$NON-NLS-2$

    System.out.println(stringPair.key); // Hello
    System.out.println(stringPair.getValue()); // World!

    GenericPair<String, Integer> stringIntegerPair = new GenericPair<>("int", new Integer(3)); //$NON-NLS-1$
    System.out.println(stringIntegerPair.key); // "int"
    System.out.println(stringIntegerPair.getValue()); // 3

    // GenericPair<String, Object> stringObjectPair = stringIntegerPair; <- this is not allowed
    GenericPair<String, Object> stringObjectPair1 = copyPair(stringIntegerPair); // this is allowed
    // GenericPair<String, Object> stringObjectPair = stringPair; <- this is not allowed
    GenericPair<String, Object> stringObjectPair2 = copyPair(stringPair); // this is allowed

    System.out.println(isSame(stringPair, stringPair)); // true
    System.out.println(isSame(stringPair, stringObjectPair1)); // true
    System.out.println(isSame(stringPair, stringObjectPair2)); // false
    // System.out.println(isSame(stringObjectPair1, stringPair)); <-- this is not allowed
    // System.out.println(isSame(stringObjectPair2, stringPair)); <-- this is not allowed
  }
}
```

We Cannot Instantiate Generic Parameters!

- We cannot instantiate generic type parameters

We Cannot Instantiate Generic Parameters!

- We cannot instantiate generic type parameters
- Now here it gets a bit tricky, listen up

We Cannot Instantiate Generic Parameters!

- We cannot instantiate generic type parameters
- Now here it gets a bit tricky, listen up
- Assume we have a class or function with the generic parameter `T`

We Cannot Instantiate Generic Parameters!

- We cannot instantiate generic type parameters
- Now here it gets a bit tricky, listen up
- Assume we have a class or function with the generic parameter `T`
- Inside this class or method, we cannot do `T x = new T();`

We Cannot Instantiate Generic Parameters!

- We cannot instantiate generic type parameters
- Now here it gets a bit tricky, listen up
- Assume we have a class or function with the generic parameter `T`
- Inside this class or method, we cannot do `T x = new T();`
- Why?

We Cannot Instantiate Generic Parameters!

- We cannot instantiate generic type parameters
- Now here it gets a bit tricky, listen up
- Assume we have a class or function with the generic parameter `T`
- Inside this class or method, we cannot do `T x = new T();`
- Why? Because of erasure.

Erasure

- Generic parameters actually only exist until the compiler has processed your code

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic...)

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters, i.e.,
  - replaces all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters, i.e.,
  - replaces all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded
  - inserts type casts if necessary to preserve type safety

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters, i.e.,
  - replaces all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded
  - inserts type casts if necessary to preserve type safety
- If you would do something like `T a = new T()`, your machine code would not have any idea what class `T` actually is

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters, i.e.,
  - replaces all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded
  - inserts type casts if necessary to preserve type safety
- If you would do something like `T a = new T()`, your machine code would not have any idea what class `T` actually is
- So it could not allocate an instance of the right type

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters, i.e.,
  - replaces all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded
  - inserts type casts if necessary to preserve type safety
- If you would do something like `T a = new T()`, your machine code would not have any idea what class `T` actually is
- So it could not allocate an instance of the right type
- It cannot even be guaranteed that a parameter-less constructor of form `T()` exists in the type actually used for `T`

Erasure

- Generic parameters actually only exist until the compiler has processed your code
- They do not exist in the produced machine code (well, in the reflection information they exist, but this is another topic. . .)
- What does this mean?
- It means that the compiler removes all generic parameters, i.e.,
  - replaces all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded
  - inserts type casts if necessary to preserve type safety
- If you would do something like `T a = new T()`, your machine code would not have any idea what class `T` actually is
- So it could not allocate an instance of the right type
- It cannot even be guaranteed that a parameter-less constructor of form `T()` exists in the type actually used for `T`
- And thus, this is not allowed

Generic Arrays

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays

Example for generic static method with generic array

Listing: Example for generic static method with generic array

```
package cn.edu.hfuu.iao;

/** a class where we use a generic array */
public class GenericsStaticFunctionsAndArrays {

  /** replace the element at index {@code index} in {@code array} with {@code replace}
   *  and return the old element that was stored there before */
  static <T> T replaceAndGetOld(T[] array, int index, final T replace) {
    T old         = array[index];
    array[index] = replace;
    return old;
  }

  /** The main routine
   *  @param args  we ignore this parameter */
  public static void main(String[] args) {
    String[] list = {"Hello ", "World, ", //$NON-NLS-1$ //$NON-NLS-2$
                     "it's ", "me."};     //$NON-NLS-1$ //$NON-NLS-2$

    for(String s : list) { System.out.print(s); }
    System.out.println(); // Hello World, it's me

    String old = replaceAndGetOld(list, 3, "someone else."); //$NON-NLS-1$
    System.out.println(old); // me.

    for(String s : list) { System.out.print(s); }
    System.out.println(); // Hello World, it's someone else.
  }
}
```

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`
  - We can do `void o(T[] a){ T[] z = a; m(z, n(z)); }`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`
  - We can do `void o(T[] a){ T[] z = a; m(z, n(z)); }`
  - But we cannot do `T[] p(int i){ return new T[i]; }`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`
  - We can do `void o(T[] a){ T[] z = a; m(z, n(z)); }`
  - But we cannot do `T[] p(int i){ return new T[i]; }`
  - And we also can never do `T x = new T();`

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`
  - We can do `void o(T[] a){ T[] z = a; m(z, n(z)); }`
  - But we cannot do `T[] p(int i){ return new T[i]; }`
  - And we also can never do `T x = new T();`
- Why?

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`
  - We can do `void o(T[] a){ T[] z = a; m(z, n(z)); }`
  - But we cannot do `T[] p(int i){ return new T[i]; }`
  - And we also can never do `T x = new T();`
- Why? Because of erasure, that's why.
- If you would do something like `T[] a = new T[3]`, your machine code would not have any idea what class `T` actually is

Generic Arrays and Erasure

- In the "Generic Map" example, it implicitly became clear that we can also use generics in arrays
- When dealing with generic arrays, erasure will make our life a bit harder
- Assume we have a class or function with the generic parameter `T`
  - We can declare and use something like `T[]`, i.e., a generic array based on generic type `T`
  - We can do `void m(T[] a, T v){ a[1] = v; }`
  - We can do `T n(T[] a){ return a[0]; }`
  - We can do `void o(T[] a){ T[] z = a; m(z, n(z)); }`
  - But we cannot do `T[] p(int i){ return new T[i]; }`
  - And we also can never do `T x = new T();`
- Why? Because of erasure, that's why.
- If you would do something like `T[] a = new T[3]`, your machine code would not have any idea what class `T` actually is
- So it could not allocate an array of the right type

Inheritance and Generics

- Actually, you already saw this, but let us explicitly mention again: You can subclass generic types

Inheritance and Generics

- Actually, you already saw this, but let us explicitly mention again: You can subclass generic types
- If you want, you can specify the generic parameters for the subclass

Example for a subclass of GenericPair

Listing: Example for a subclass of GenericPair

```
package cn.edu.hfuu.iao.collections;

/**
 * a generic key-value pair where we can specify the key type but have String values
 * @param <K>
 *          the generic key type
 */
public class StringValuedPair<K> extends GenericPair<K, String> {

  /** create */
  public StringValuedPair(final K _key, final String _value) {
    super(_key, _value);
  }

  /** get the value */
  @Override
  public String getValue() {
    return '\'' + super.getValue() + '\'';
  }
}
```

Example for using the new String-valued Pair

Listing: Example for using the new String-valued Pair

```
package cn.edu.hfuu.iao;

import cn.edu.hfuu.iao.collections.GenericPair;
import cn.edu.hfuu.iao.collections.StringValuedPair;

/** a class where we use the string-valued generic Pair class */
public class StringValuedPairTest {
  /** The main routine
   *  @param args we ignore this parameter */
  public static void main(String[] args) {
    GenericPair<String,String> stringPair = // StringValuedPair is compatible
            new StringValuedPair<String>("Hello", //$NON-NLS-1$
                                         "World!"); //$NON-NLS-1$

    System.out.println(stringPair.key); // Hello
    System.out.println(stringPair.getValue()); // 'World!'
  }
}
```

Summary

- Generics allow us to specify placeholders for types in a class implementation
- When instantiating the class, we then determine the actual types
- This provides additional type safety while allowing us to implement and use very general base classes that apply to arbitrary types
- And it reduces the number of explicit type casts we need to do
- Generics can also be applied to methods
- We can define lower bounds for generic type parameters via `<T extends MyObject>`
- We can use wildcards `?` for generic type parameters
- We have learned what erasure is and that we cannot instantiate generic parameters or arrays thereof.

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China

Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog