KATHOLIEKE UNIVERSITEIT LEUVEN

Department of Computer Science

Generic Classes

- Definition and instantiation
- Wildcards
- Generic methods

KATHOLIEKE UNIVERSITEIT LEUVEN

Assignment

- Parameterize a given hierarchy of bounded array lists
  - The hierarchy involves a top class of bounded array lists, complemented with a subclass of ordered bounded array lists
    - Bounded array lists accept and return elements of static type "Object"
    - Elements stored in ordered bounded array lists must be comparable
  - Bounded array lists check at runtime whether their elements are of the proper type

Overview

KATHOLIEKE UNIVERSITEIT LEUVEN

- Definition of generic types
- Polymorphism
- Generic methods
- Inheritance

Definition

KATHOLIEKE UNIVERSITEIT LEUVEN

- The definition of a generic class or a generic interface involves one or more **type parameters**
  - Formal type parameters can only be bound to reference types, not to primitive types
    - In Java, generic classes and generic interfaces cannot have other kinds of arguments, such as integers, …
- Formal type parameters can be **used** (almost) everywhere a type can be used
  - No creation of objects of formal parameter type (`new S(…)`)
    - No creation of arrays of formal parameter type (`new S[10]`)

```
public class GenericClass<S,T> {
  public S someMethod(T p) {
    S localVar;
    …
  }
}
```

Instantiation

KATHOLIEKE UNIVERSITEIT LEUVEN

- A generic class is **instantiated** by supplying actual reference types for all its formal arguments
  - An actual argument in instantiating a generic class may in turn be an instantiated generic class
- With generic classes, the Java compiler can **perform checks** that had to be done at run-time otherwise
  - Actual arguments must have the proper type
  - Lots of type casts are avoided

```
GenericClass<Integer,Person> myVariable =
   new GenericClass<Integer,Person>(…);

//Arguments for someMethod must have static type Person!
//Type cast needed here in non-generic code!
Integer myResult =
   myVariable.someMethod(new Person());
```

Compilation

KATHOLIEKE UNIVERSITEIT LEUVEN

- A generic class is **compiled once and for all**, resulting in a single class file
  - All objects of instantiations of a generic class belong to the same class
- The class underlying all instantiations of a generic class is called a **raw type**
  - For reasons of backward compatibility, raw types may still be used in programs (Oracle deprecates their usage in new code)

Task 1

```
// The raw type for GenericClass
public class GenericClass {
  public Object someMethod(Object p) {
    Object localVar;
    …
  }
}
// Declaration of a variable of raw type.
GenericClass myRawObject = …;
```

Polymorphism

KATHOLIEKE UNIVERSITEIT LEUVEN

- Instantiations of a generic class with different actual types **never have a subtype/supertype relation**
  - This is true even if the one actual type is a subclass of the other actual type
    - `Integer` is a subtype of `Number`;
    - `List<Integer>` is not a subtype of `List<Number>`
- This rule does not apply to built-in arrays
  - `Integer[]` is a subtype of `Number[]`;
    - Arrays of the former type are thus assignable to variables of the latter type
    - `ArrayStoreException` is thrown if, via polymorphism, inappropriate elements are registered in an array

Task 2

Wildcards

KATHOLIEKE UNIVERSITEIT LEUVEN

- Java offers **wildcards** to approach generic objects in a polymorphic way
  - The most general wildcard stands for any type, and is denoted `?`
    - `List<?>` is the supertype of all instantiations of the generic class `List`
- Methods expecting arguments of a formal generic type can not be invoked against variables of wildcard-type
  - Given `List<?> myList`, `myList.add(new Integer())` is invalid
    - There is no guarantee that the actual argument is of proper type
- Methods returning an object of formal generic type can be invoked against variables of wildcard-type
  - Given `List<?> myList`, `Object o = myList.get(0)` is valid
    - The result can be safely assigned to a variable of type `Object`

Task 3

Wildcards: Upper Bound

KATHOLIEKE UNIVERSITEIT LEUVEN

- A wildcard with upper bound `T` stands for type `T` or any sybtype of `T`
  - A wild card with upper bound `T` is denoted "`? extends T`"
    - `List<? extends Number>` is the supertype of all instantiations of generic class `List` involving `Number` or a subclass of `Number`
- Restrictions that apply to unrestricted wildcards equally apply to bounded wildcards with upper bound
  - Methods expecting arguments of a formal generic type can not be invoked
    - Given `List<? extends Number>, myList, myList.add(new Integer())` is invalid
  - Methods returning an object of formal generic type can be invoked against variables of wildcard-type
    - Given `List<? extends Number>, myList, Number o = myList.get(0)` is valid

Wildcards: Lower Bound

KATHOLIEKE UNIVERSITEIT LEUVEN

- A wildcard with lower bound `T` stands for type `T` or any supertype of `T`
  - A bounded wild card with lower bound `T` is denoted "`? super T`"
    - `List<? super Number>` is the supertype of all instantiations of generic class `List` involving `Number` or a superclass of `Number`
- Restrictions
  - Methods expecting arguments of a formal generic type can be invoked, if the actual argument does not exceed the lower bound
    - Given `List<? super Integer>`
      - `myList.add(new Integer())` is valid
      - `myList.add(new Object())` is not valid
  - Methods returning an object of formal generic type can be invoked against variables of wildcard-type
    - Given `List<? super Integer>, myList, Object o = myList.get(0)` is valid

Task 4

Subclassing

KATHOLIEKE UNIVERSITEIT LEUVEN

- A generic type can be defined to inherit from **an instantiated generic type**
  - The generic subtype typically inherits from its generic supertype instantiated with its own formal arguments
    - Formal type arguments may use upper bounded wildcards
  - A generic type can also be defined to inherit from **an ordinary class** or to implement **an ordinary interface**
- Types (generic or non-generic) cannot be defined to inherit from a **generic type**

Task 5

```
public class GenericSubClass<S extends Number,T>
    extends GenericSuperclass<S>
    implements GenericInterface<T>
{
  …
}
```

Generic Methods: Definition

KATHOLIEKE UNIVERSITEIT LEUVEN

- A generic method is **parameterized** in one or more types
  - The generic parameters precede the return type of the method
    - Generic instance methods are exceptional: instance methods typically use the parameters of the generic class to which they belong
- Generic arguments are used to express dependencies among types of formal arguments involved in a method
  - Bounded wildcards in both directions are used for that purpose

```
public <T> void genericMethod(Collection<T> coll)
    { … }

public static <T> void genericMethod
        (Collection<T> coll, Iterator<? super T>)
    { … }
```

Generic Methods: Invocation

KATHOLIEKE UNIVERSITEIT LEUVEN

- Generic methods can be invoked by supplying actual types for their generic arguments
  - Actual generic arguments are specified in front of the name of the invoked method
- The actual type(s) of a generic method can also be **derived from the types of its actual arguments**
  - The compiler rejects invocations if no matching type is available

Task 6

```
Collection<Integer> theCollection = …;
Iterator<Number> theIterator = …;

TheClass.<Integer> genericMethod
    (theCollection, theIterator);

TheClass.genericMethod(theCollection, theIterator);
```