Object Oriented Programming

Java

Part 8: Generic types

Generic types – revision

- A **generic class** (or **parametric class**) is a class that receives as argument other classes. Instances of **generic classes** are denominated **parameterized classes**.
- Generic classes are commonly used to define collections (sets, lists, queues, trees, etc).

Motivation

- Before generics
  (version 4 and previous versions):

```
List v = new ArraysList();
v.add(“test”);
Integer i = (Integer)v.get(0);
```

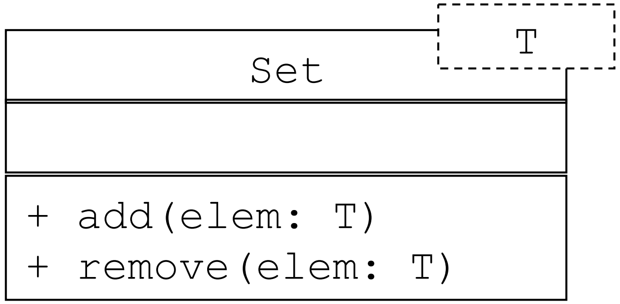
Compile without errors, but throws an exception (in run-time) in the 3rd line!

- After generics
  (version 5 and following versions):

```
List<String> v = new ArraysList<String>();
v.add(“test”);
Integer i = v.get(0);
```

Compile-time error in the 3rd line ☺

Generic types (1)



T

Set

+ add(elem: T)
+ remove(elem: T)

```
public class Set<T> {
     public void add(T elem) {...}
     public void remove(T elem) {...}
}
```

Generic types (2)

```
public class Element<T> {
     protected T element;
     protected Element<T> next;
     public Element(T element) {
          this.element = element;
     }
     public Element(T element, Element<T> next) {
          this.element = element;
          this.next = next;
     }
}
```

- To reference/build an object of type `Element` a **generic type invocation** must be perfomed, which replaces `T` with some concrete value, such as `String`:

```
Element<String> strElem = new Element<String>("Hello");
```

Generic types (3)

```java
public class Set<T> {
    protected int nbElems;
    protected Element<T> head;
    public void add(T elem) {
        //verify if the element already exist in the set
        for (Element<T> aux=head; aux!=null; aux=aux.next)
            if (elem.equals(aux.element)) return;
        //if not, add it in the beginning
        head =  new Element<T>(elem,head);
        nbElems++;
    }
    public void remove (T elem) {...}
}
```

Generic types (4)

- Once again, to reference/build an object of type `Set` it is necessary to perform a generic type invocation. For instance, a set of object of type `String` is referenced/ built like:

```
Set<String> set = new Set<String>();
```

- Adding objects of type `String` to the set would be:

```
set.add(“Hello”);
set.add(“World”);
set.add(“!”);
```

Generic types (5)

- Declaring a variable `set` of type `Set<String>` tells the compiler that `set` is a reference to an object of type `Set<T>` where `T` is a `String`.
  - It is not created a class `Set<String>`.
  - The types `Set<String>` and `Set<Number>` are not two distinct classes.
- As for methods, in the generic types we also have the concept of parameter/argument:
  - In the context of `Set<T>`, `T` is said to be a **(type) parameter**.
  - In the context of `Set<String>`, `String` is said to be a **(type) argument**.

Generic types (6)

```
public class Set<T> {
     protected int nbElems;
     protected Element<T> head;
     private static int nbNextSet=0;
     private int nbSet;
     public Set() {
           nbSet = nbNextSet++;
     }
     public int nbSet() {
          return nbSet;
     }
     public static int nbNextSet() {
          return nbNextSet;
     }
     public void add(T elem) {...}
     public void remove (T elem) {...}
}
```

Generic types (7)

```
Set<String> set_s = new Set<String>();
System.out.println(“set_s has number ” + set_s.nbSet());
Set<Integer> set_i = new Set<Integer>();
System.out.println(“set_i has number ” + set_i.nbSet());
```

In the terminal is printed

```
set_s has number 0
set_i has number 1
```

Generic types (8)

- Consequences:
  - The type parameter `T` cannot be used in static contexts.
  - The access to static members of the parameterized classes cannot be done through the parameterized type:

```
Set<String>.nbNextSet(); //INVALID!!!
```

```
Set.nbNextSet();
```

Generic types (9)

- It is not possible to use the type parameter `T` to build objects (nor to build arrays).

```
public class Set<T> {
     // ...
     public T[] convertSetToArray() {
          T[] res = new T[nbElems]; //INVALID!!!
          //copy the elements from this set to the array
     }
}
```

Generic types (10)

- Solution 1: pass the array as a parameter to the method `convertSetToArray` which fills this array with the elements of the set.

```
public class Set<T> {
     // ...
     public T[] convertSetToArray(T[] array) {
          int i = 0;
          for (Element<T> aux=head; aux!=null; aux=aux.next)
               array[i++] = aux.element;
          return array;
     }
}
```

Generic types (11)

- When defining a generic type it is possible to restrict the type argument that is passed to the type parameter `T`:

```
interface OrderedCollection<T extends Comparable<T>> {...}
```

  - The argument type passed to the type argument `T` implements the methods of the interface `Comparable<T>`.

```
interface CharSequenceOrderedCollection<T extends
                                Comparable<T> & CharSequence> {...}
```

  - The argument type passes to the parameter type `T` implements the methods of the interface `Comparable<T>` and implements the methods of the interface `CharSequence`.

Generic types (12)

- The keyword `extends` is used in the type parameters of the generic types in a very general form:
  - It means `extends` if the type that follows is a class.
  - It means `implements` if the type that follows is an interface.

Parameterized types (1)

- The generic types might be used in the context of the declaration/instantiation of parameterized types:
  - Types of fields;
  - Types of local variables;
  - Types of method parameters;
  - Type of method return.

Parameterized types (2)

- Consider a method that sum the elements in a `Set<Number>`:

```
static double sum(Set<Number> set) {
     double res = 0.0;
     for (Element<Number> aux=set.head; aux!=null; aux=aux.next)
          res+=aux.element.doubleValue();
     return res;
}
```

Parameterized types (3)

- If the `sum` method is invoked from a `Set<Integer>`, the code does not compile:

```
Set<Integer> set = new Set<Integer>();
set.add(1);
set.add(2);
double sum = sum(set); //INVALID!!!
```

- Although `Integer` is a subtype of `Number`, `Set<Integer>` is not a subtype of `Set<Number>`.

Parameterized types (4)

- The solution is to define the parameter of method `sum` as a set of `Number` or of any subtype of `Number`:

```
static double sum(Set<? extends Number> set) {
     double res = 0.0;
     for (Element<? extends Number> aux=set.head;
               aux!=null; aux=aux.next)
          res+=aux.element.doubleValue();
     return res;
}
```

- The `? extends` in the type parameter refers to a `Number` or to a subtype of `Number`.

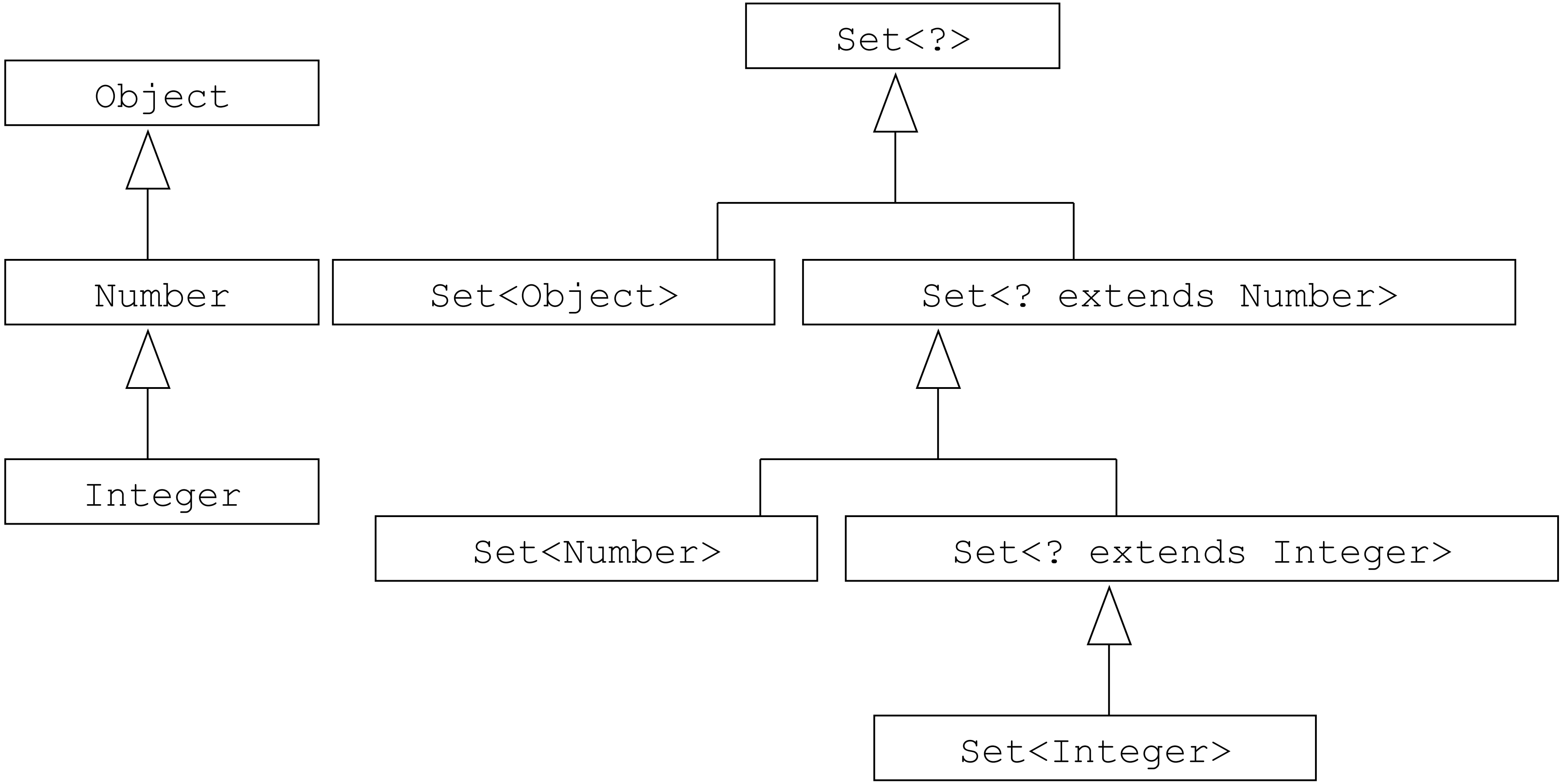
Parameterized types (5)

- It is also possible to define the type parameter as being of a certain type or of any supertype of that type.
  - `Set<? super Integer>` denotes the set of `Integer`, or of a supertype of `Integer`:
    - `Set<Integer>`
    - `Set<Number>`
    - `Set<Object>`
    - …
- The `extends` and the `super` cannot be used simultaneously.

Parameterized types (6)

- It is also possible to define the type parameter as being of any type.
  - The `Set<?>` denotes a set of any type.
  - Implicitly, `Set<?>` refers to any type provided that it is an `Object` or any subtype of it.
    - The `Set<?>` is another way of writing `Set<? extends Object>` (and not `Set<Object>`).

Parameterized types (7)



Parameterized types (8)

- As `?` represents an unknown type, it is not possible to invoke any method of the parameterized type that receives the type parameter as parameter of the method:

```
Set<?> set = new Set<Integer>();
set.add(1); //INVALID!!!
```

```
Set<? extends Integer> set = new Set<Integer>();
set.add(1); //INVALID!!!
```

```
Set<? super Integer> set = new Set<Integer>();
set.add(1);
set.add(new Integer(2));
```

```
Set<? super Integer> set = new Set<Integer>();
set.add(1.0); // INVALID!!!
set.add(new Object()); //INVALID!!!
```

Parameterized types (9)

```
Set<?> set = new Set<Integer>();
System.out.println(set.nbSet());
```

```
Set<? extends Integer> set = new Set<Integer>();
System.out.println(set.nbSet());
```

```
Set<? super Integer> set = new Set<Integer>();
System.out.println(set.nbSet());
```

Parameterized types (10)

- The `?` can only be used in the context of the declaration of parameterized types:
  - Types of fields;
  - Types of local variables;
  - Types of method parameters;
  - Type of method return.

Limits of type parameters and type arguments (1)

- **(Type) Parameter**
  (in the definition of generic methods and types)

```
void foo(int n, char c) {...} //n and c are parameters
class Set<T> {...}            //T is a type parameter
<T> T pickout() {...}         //T is a type parameter
```

- **(Type) Argument**
  (in the invocation of generic methods or declaration/ instantiation of parameterized types)

```
foo(5,‘a’);                   //5 and ‘a’ are arguments
Set<?> set1;                  //? is a type argument
set1 = new Set<String>;       //String is a type argument
this.<String>pickout();       //String is a type argument
```

Limits of type parameters and type arguments (2)

- In the definition of generic types/methods, a type parameter of the form:
  - `<T>` is said to be a **plain type parameter**.
  - `<T extends Number>` is said to be an **upper-bounded type parameter** (in this case `Number`).
- In the declaration of parameterized types, a type argument of the form:
  - `<T>` is said to be a **plain type argument**.
  - `<T extends Number>` or `<? extends Number>` is said to be an **upper-bounded type argument** (in this case `Number`).
  - `<T super Number>` or `<? super Number>` is said to be a **lower-bounded type argument** (in this case `Number`).
  - `<?>` is called an **unlimited type argument**.

Generic methods and constructors (1)

```
public class Set<T> {
     // ...
     public T[] convertSetToArray(T[] array) {
          //copy the elements of this set to the array
          return array;
     }
}
```

- The method `convertSetToArray` as it is defined is **too restrictive** (see slide 11 and 12):
  - In an object of type `Set<Integer>` we need to pass as parameter an array of `Integer[]`.
  - In an object of type `Set<Integer>` we cannot pass as parameter an array of `Object[]`, even if it is valid to store an object of type `String` in such an array.

Generic methods and constructors (2)

- A **generic method** is declared by defining the type parameter between the modifiers and the return type of the method:

```
public class Set<T> {
    //...
    public <E> E[] convertSetToArray(E[] array) {
        Object[] tmp = array;
        int i = 0;
        for (Element<T> aux=head; aux!=null; aux=aux.next)
            tmp[i++] = aux.element;
        return array;
    }
}
```

Generic methods and constructors (3)

```
Set<Integer> set = new Set<Integer>();
Object[] array = new Object[set.nbElems];
array = set.convertSetToArray(array);
```

```
Set<Integer> set = new Set<Integer>();
Integer[] array = new Integer[set.nbElems];
array = set.convertSetToArray(array);
```

Generic methods and constructors (4)

- A generic method or constructor does not need to be declared within a generic class, but if it is the type parameters are different.

Generic methods and constructors (5)

- The method `convertSetToArray` can be invoked as:

```
Set<Integer> set = new Set<Integer>();
Integer[] array = new Integer[set.nbElems];
array = set.<Integer>convertSetToArraya(array);
```

- This **parameterized invocation** tells the compiler that the type parameter `E` of the method `convertSetToArray` must be treated as an `Integer`, and that the arguments and the return type must comply with that.

Generic methods and constructors (6)

- The parameterized invocation is rarely needed. The compiler is able, in general, to perform the **type inference**:

```
Set<Integer> set = new Set<Integer>();
Integer[] array = new Integer[set.nbElems];
array = set.convertSetToArray(array);
```

- The type inference is based on the **static type** of the argument passed to the generic method or constructor (and not on its dynamic type).
  - Declaration type of explicit cast.

Generic methods and constructors (7)

- The parameterized invocation must be always done through a qualified name:
  - `this.<Type>method(params);`
  - `super.<Type>method(params);`
  - `ref.<Type>method(params);`
  - `IdentC.<Type>method(params);`
    to static methods, where `IdentC` is the name of the respective class.

```
String s1 = “Hello”;
String s2 = <String>passObject(s1); //INVALID!!!
```

Generic methods and constructors (8)

```
<T> T passObject(T obj) {
       return obj;
}
```

```
String s1 = “Hello”;
String s2 = this.<String>passObject(s1);
```

```
String s1 = “Hello”;
String s2 = passObject(s1);          // T -> String
Object o1 = passObject(s1);          // T -> String
Object o2 = passObject((Object)s1);  // T -> Object
```

```
String s1 = “Hello”;
s1 = passObject((Object)s1); //INVALID!!!
```

```
String s1 = “Hello”;
s1 = (String) passObject((Object)s1); // T -> Object
```

Subtypes and generic types (1)

- It is possible to:
  - Define a subtype (generic or not) from a non-generic supertype.
  - Define a subtype (generic or not) from a generic supertype.

```
class GeneralList<E> implements List<E> {...}
class StringList implements List<E> {...}
class NumberSet<T extends Number> extends Set<T> {...}
class IntegerSet extends Set<T> {...}
...
```

Subtypes and generic types (2)

- A class cannot implement two interfaces which are different parameterizations of the same interface.

```
class Value implements Comparable<Value> {...}
class ExtendedValue extends Value
    implements Comparable<ExtendedValue> {...} //INVALID!!!
```

- In this case, a class `ExtendedValue` should implement both interfaces `Comparable<Value>` and `Comparable<ExtendedValue>` (which is not allowed).

Erasure (1)

- For each parameterized type there is only one type.
  - Which type is this?
  - Given the generic type `Set<T>`, what type do `Set<Integer>` and `Set<Number>` share?
- The compiler erases all information about the parameter type in the compiled type:
  - The `Set<T>` is in the compiled type only `Set`.
  - The parameter type `T` is in the compiled type:
    - `Object`, when it is found in a `<T>` context (`Object` is the implicit superclass of `T`).
    - `Number`, when it is found in a `<T extends Number>` context (`Number` is the explicit superclass of `T`).

Erasure (2)

- The information erased from the generic type is called **erasure**.
  - `Set` is the erasure of `Set<T>`.
  - `Object` is the erasure of `T` in a `<T>` context.
  - `Number` is the erasure of `T` in a `<T extends Number>` context.
- The erasure of a generic type is also called the **raw type**.
  - `Set` is the raw type of `Set<T>`.
- The compiler generates a class definition for the erasure of a generic type by effectively replacing each type variable with its erasure.

Erasure (3)

```
public class PassObject<T> {
     T passObject(T t) { return t; }
}
```

```
public class PassObject {
     Object passObject(Object t) { return t; }
}
```

- When a parameterized type is used and the type information from the erasure of the generic type does not match what is expected, the compiler inserts a cast.

```
PassObject<String> pos = new PassObject<String>();
String s1 = “Hello”;
s1 = pos.passObject (s1);
```

```
PassObject pos = new PassObject();
String s1 = “Hello”;
s1 = (String) pos.passObject(s1);
```

Erasure (4)

```
public class Element<T> {
     protected T element;
     protected Element<T> next;
     public Element(T element) {...}
     public Element(T element, Element<T> next) {...}
}
```

```
public class Element {
     protected Object element;
     protected Element next;
     public Element(Object element) {...}
     public Element(Object element, Element next) {...}
}
```

Erasure (5)

```
public class Set<T> {
     protected int nbElems;
     protected Element<T> head;
     public void add(T elem) {...}
     public void remove (T elem) {...}
     public T pickout() {
          return head.element;
     }
}
```

```
public class Set {
     protected int nbElems;
     protected Element head;
     public void add(Object elem) {...}
     public void remove(Object elem) {...}
     public Object pickout() {
          return head.element;
     }
}
```

Erasure (6)

```
Set<String> setstr = new Set<String>();
String s1 = “Hello”;
setstr.add(s1);
s1 = setstr.pickout();
```

```
Set setstr = new Set();
String s1 = “Hello”;
setstr.add(s1);
s1 = (String) setstr.pickout();
```

Erasure (7)

- Because of erasure it is not allowed, at runtime, anything that requires knowledge of the type argument:
  1. You cannot use the type parameter `T` to create objects or to create arrays.

```
public class Set<T> {
     // ...
     public T[] convertSetToArray() {
          T[] res = new T[nbElems]; //INVALID!!!
          //copy the elements from this set to the array
     }
}
```

Erasure (8)

2. It is not allowed to create arrays whose elements are parameterized types, with the exception of parameterized types with `?`.

```
Set<String>[] tcjs = new Set<String>[2]; //INVALID!!!
```

```
Set<?>[] array = new Set<?>[2];
array[0] = new Set<String>();
array[1] = new Set<Integer>();
((Set<String>)array[0]).add("Hello");
((Set<Integer>)array[1]).add(1);
System.out.println(array[0].pickout());
System.out.println(array[1].pickout());
```

Erasure (9)

3. It is not allowed to use the `instanceof` to verify if one object is an instance of a parameterized type, with the exception of types parameterized with `?`.

```
Set<String> strings = new Set<String>();
boolean b = strings instanceof Set<?>;
```

```
boolean b = strings instanceof Set<String>; //INVALID!!!
```

```
Set<?> strings = new Set<String>();
System.out.println(strings instanceof Set<?>);
```

```
System.out.println(strings instanceof Set<String>); //INVALID!!!
```

Erasure (10)

4. Casts involving parameterized types or type parameters are replaced by casts for the corresponding erasures.
   - Usually, this causes the compiler to generate **unchecked warnings**: **the cast can not be verified at runtime, nor guaranteed to be safe at compile time.**

Erasure (11)

```
void addStringHello(Set<?> set) {
     Set<String> strings = (Set<String>) set; //unchecked
     strings.add(“Hello”);
}
```

```
void addStringHello(Set set) {
     Set strings = (Set) set;
     strings.add(“Hello”);
}
```

```
Set<Integer> integers = new Set<Integer>();
addStringHello(integers);          //ok
Integer nb = integers.pickout(); //runtime error!!!
```

```
Set integers = new Set();
addStringHello(integers);
Integer nb = (Integer) integers.pickout();
```

Erasure (12)

```
Object passObject(Objecto obj) {
    return obj;
}
```

```
Set<String> strings = new Set<String>();
Object obj = passObject(strings);
Set<String> cj1 = (Set<String>) obj;  // unckecked
Set<String> cj2 = (Set) obj;          // unckecked and raw type
Set<?> cj3 = (Set) obj;               // ok but raw type
Set<?> cj4 = (Set<?>) obj;            // ok
```

- The raw types exist for legacy reasons and they should be avoided.

Overloading in generic types (1)

- In Java, the **signature of a method** is defined by:
  - Identifier of the method.
  - Number and type of the parameters of the method.
- The **signature of a generic method** is defined by:
  - Identifier of the method.
  - Number and type of the parameters of the method.
  - Number and upper-bound of the type parameters of the method.
- Two methods have an **override-equivalent signature** if they have the same signature, or if after erasure they have the same signature.
- A method is said to be an **overload** of other method if both have the same identifier but do not have override-equivalent signatures.

Overloading in generic types (2)

```
class SuperClass<T> {
     void m(int x) {}
     void m(T t) {}
     void m(String s) {}
     <N extends Number> void m(N n) {}
     void m(Set<?> cj) {}
}
```

- The class `SuperClass` defines five overloads of the method `m`:
  - `void m(int x) {}`
  - `void m(Object t) {}`
  - `void m(String s) {}`
  - `void m(Number n) {}`
  - `void m(Set cj) {}`

Overloading in generic types (3)

- It is an error a class or interface declare two methods with the same identifier and the same signature after erasure.
- Defining any of the following methods in `SuperClass` should be an error:
  - `void m(Object o) {}`
  - `void m(Number n) {}`
  - `<G extends String> void m(G g) {}`
  - `void m(Set<Object> cj) {}`

```
class SuperClass<T> {
        void m(int x) {}
        void m(T t) {}
        void m(String s) {}
        <N extends Number> void m(N n) {}
        void m(Set<?> cj) {}
}
```

Overriding in generic types (1)

- A method in a subtype is an **redefinition** of a method in a supertype if:
  1. Both methods have override-equivalent signatures, with the following restriction:
     - The signature of the method of the subtype must be the same as the one of the method of the supertype, or must be the same after erasure of the signature of the method of the supertype.
  2. The return of the methods must be covariant (the return of the method of the subtype must be the same or a subtype of the return of the method of the supertype).
- The condition 1. implies:
  - A generic method cannot redefine a non-generic method.
  - A generic method can redefine a generic method.
  - A non-generic method can redefine a generic method.

Overriding in generic types (2)

```
class SuperClass<T> {
     void m(int x) {}
     void m(T t) {}
     void m(String s) {}
     <N extends Number> void m(N n) {}
     void m(Set<?> cj) {}
}
```

```
class SuperClass {
     void m(int x) {}
     void m(Object t) {}
     void m(String s) {}
     void m(Number n) {}
     void m(Set cj) {}
}
```

```
class SubClass<T> extends SuperClass<T> {
     void m(Integer i) {}
     void m(Object t) {}
     void m(Number s) {}
}
```

Overriding in generic types (3)

- Relatively to the previous example:
  - The method `m(Integer i)` is an overloading of the `m` methods of `SubClass`.
  - The method `m(Object t)` of `SubClass` is a redefinition of the method `m(T t)` of `SuperClass`.
  - The method `m(Number cj)` of `SubClasse` is a redefinition of `m(N n)` of `SuperClass`.

Overriding in generic types (4)

```
class SuperClass<T> {
     void m(int x) {}
     void m(T t) {}
     void m(String s) {}
     <N extends Number> void m(N n) {}
     void m(Set<?> cj) {}
}
```

```
class SuperClass {
     void m(int x) {}
     void m(Object t) {}
     void m(String s) {}
     void m(Number n) {}
     void m(Set cj) {}
}
```

```
class SubClass<T> extends SuperClass<T> {
     void m(Number n) {}                    //ok
     <S extends String> void m(S s) {}  //INVALID!!!
}
```

```
class SubClass extends SuperClass {
     void m(Number n) {}
     void m(String s) {}
}
```

Overriding in generic types (5)

- Relatively to the previous example:
  - The method `m(Number n)` of the `SubClass` is a redefinition of the method `m(N n)` of the `SuperClass`.
  - The method `m(S s)` of the `SubClass` tries to make a redefinition of the method `m(String s)` of the `SuperClass`, but it is not allowed!

Overriding in generic types (6)

```
class SuperClass {
     protected Integer i;
     <N extends Number> Number m1(N n) {return i;}
     <N extends Number> N m2() {return (N) i;}
}
```

```
class SuperClass {
     protected Integer i;
     Number m1(Number n) {return i;}
     Number m2() {return (Number) i;}
}
```

```
class SubClass extends SuperClass {
     <N extends Number> Integer m1(N n) {return i;}
     <N extends Integer> N m2() {return (N) i;}
}
```

```
class SubClass extends SuperClass {
     Integer m1(Number n) {return i;}
     Integer m2() {return (Integer) i;}
}
```

Overriding in generic types (7)

- Relatively to the previous example:
  - The method `Integer m1(N n)` of the `SubClass` is a redefinition of the method `Number m1(N n)` of the `SuperClass`.
  - The method `N m2()` of the `SubClass` is not a redefinition of the method `N m2()` of the `SuperClass`.

Overriding in generic types (8)

```
class SuperClass {
     protected Integer i;
     <N extends Number> Number m1(N n) {return i;}
     <N extends Number> N m2() {return (N) i;}
}
```

```
class SubClass extends SuperClass {
     @Override //ok
     <N extends Number> Integer m1(N n) {return i;}
     @Override //INVALID!!!
     <N extends Integer> N m2() {return (N) i;}
}
```