ADVANCED PROGRAMING PRACTICES

Java generics

Generic programming: introduction

- Java Generics is a language feature that enables the definition of classes and methods that are implemented independently of some type that they use as an **abstraction** by accepting a type parameter.
- The goal is to define types and algorithms that apply in **different contexts**, but where the interface and general functioning and implementation can be defined such that it **applies independently** of the context of application.
- Generic types are **instantiated** to form **parameterized types** by providing actual type arguments that replace the formal type parameters.

- For example, a class `Stack<T>` is a generic type that has a type parameter `T`. Instantiations, such as `Stack<String>` or a `Stack<Integer>`, are called **parameterized types**, and `String` and `Integer` are the respective actual **type arguments**.

Generics: history

- M.D. McIlroy. *Mass-Produced Software Components*, Proceedings of the 1st International Conference on Software Engineering, Garmisch Partenkirchen, Germany, 1968.
- Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. *Abstraction mechanisms in CLU*. Commun. ACM 20, 8 (August 1977), 564-576. doi=10.1145/359763.359789
- Joseph A. Goguen. *Parameterized Programming*. IEEE Trans. Software Eng. 10(5) 1984.
- David R. Musser, Alexander A. Stepanov. *Generic Programming*. In International Symposium on Symbolic and Algebraic Computation (ISSAC 1988). Lecture Notes in Computer Science 358, Springer-Verlag, 1989, pp 13-25.

- 1999: Sun Microsystems proposes to add generics to Java, based on GJ.
- 2001: Sun Microsystems releases a prototype including Java Generics.
- 2003: Java Generics included in Java 1.5.

Java generics: implementation

- The Java compiler uses a technique called **type erasure** to translate generic Java classes into executable bytecode.
  - Type erasure eliminates all generic type information at **compile time**.
  - All the type information between angle brackets is removed so, for example, a parameterized type like `Stack<String>` is converted into a `Stack` **raw type**.
  - All remaining uses of type variables are replaced by the upper bound of the type variable (or `Object` if there is no type bound).
  - Whenever the resulting code isn't type-correct, a cast to the appropriate type is automatically inserted.

Java generics vs. C++ templates

- While Java generics syntactically look like C++ templates and are used to achieve the same purpose, it is important to note that they are not implemented using the same concepts, nor do they provide the same programming features.
- Java generics simply provide compile-time type safety and eliminate the need for explicit casts when using type-abstract types and algorithms.
- Java generics use a technique known as **type erasure**, and the compiler keeps track of the generic definitions internally using a raw type, hence using the same class definition at compile/run time.
- A C++ template on the other hand use **template metaprogramming**, by which whenever a template is **instantiated** with a new type parameter, the entire code for the template instance is generated adapted to the type parameter and then compiled, hence having several definitions for each template instance at run time.

Generic classes/methods: definition

- A class or method that is defined with a parameter for a type is called a generic class/method or a parameterized class/method
  - For classes, the type parameter is included in angular brackets after the class name in the class definition heading.
  - For methods, the type parameter is included before the method definition.
  - Methods can define/use additional type parameters additional to their class' type parameters.
  - The type parameters are to be used like other types used in the definition of a class/method.
  - When a generic class is used, the specific type to be used is provided in angular brackets.
  - When a generic method is called, its call's parameter/return type are used.

```java
/**
 * Generic class that defines a wrapper class around a single
 * element of a generic type.
 */
public class Box<T extends Number> {

    private T t;

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

    public T get() {
        return t;
    }
/**
 * Generic method that uses the generic type of the class
 * it belongs to, that is bound to the Number type.
 */
    public void inspect(){
        System.out.println("T: " + t.getClass().getName());
    }

    public <U> void inspectWithAdditionalType(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect();
        integerBox.inspectWithAdditionalType("Hello world");
        Integer i = integerBox.get();
    }
}
```

Generic classes/methods: type erasure

- When the class is compiled, **type erasure** is applied on the type parameter **for each specific use** of the generic class or method:
  - Every occurrence of the type parameter is replaced with the highest type applicable to the type parameter.
  - If a type bound was specified, this type is applied. If no type bound was specified, `Object` is used.
  - If a value is extracted from a generic class or returned from a generic method that is of the type parameter type, its type is automatically casted to the type used at instantiation.
  - The resulting code is now a regular Java class.

```java
/**
 * Generic class that defines a wrapper class around a single
 * element of a generic type.
 */
public class Box {

    private Number t;

    public void set(Number t) {
        this.t = t;
    }

    public Number get() {
        return t;
    }
/**
 * Generic method that uses the generic type of the class
 * it belongs to, that is bound to the Number type.
 *
 */
    public void inspect(){
        System.out.println("T: " + t.getClass().getName());
    }

    public void inspectWithAdditionalType(Object u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box integerBox = new Box();
        integerBox.set(new Integer(10));
        integerBox.inspect();
        integerBox.inspectWithAdditionalType("Hello world");
        Integer i = (Integer)integerBox.get();
    }
}
```

Generic classes: benefit

- So, what is the difference between a generic class and a class defined using `Object` as the internal type? Consider a `LinkedList` class that can contain elements of type `Object`:

```
LinkedList list = new LinkedList();
list.add("abc");                // fine
list.add(new Date());           // fine as well
```

- This seems interesting, until we get the elements from the list:

```
String s = (String)list.get(0);  // cast required
Date d   = (Date)list.get(1);    // cast required
```

- As the elements are of type `Object`, we must explicitly cast them to use them as objects of their own type after extraction.
- Do you see any problem with that?

Generic classes: benefit

- The problem is that the compiler cannot check at compile time whether such casts are valid or not. Upon execution, depending on what was actually stored as the elements of the list, the runtime system might throw a `ClassCastException` if the explicit casts are invalid.
- Using generic classes, we can define such a `LinkedList` and parameterize it for every specific use and ensuring **type safety** for each different use of the generic class:

```
LinkedList<String> stringList = new LinkedList<String>();
stringList.add("Hello");                // fine
// list.add(new Date(1,1));             // error
String s = stringList.get(0);           // no cast needed

LinkedList<Integer> integerList = new LinkedList<Integer>();
integerList.add(new Integer(10));       // fine
// integerList.add("Hello");            // error
Integer i = integerList.get(0);         // no cast needed
```

- Thus, generic classes and the type erasure mechanism allow the programmer to:
  - Define classes that are valid in different contexts of use.
  - Ensure that they are used correctly in each specific context of use.

Generics: parameter type bounds

- When **defining** a generic class/method, a **type bound** can be stated on any type parameter.
- A type bound can be any **reference type** (i.e. a non-basic type).
- It **restricts** the set of types that can be used as type arguments and gives access to the non-static methods of the type it mentions.
  - A type parameter can be unbounded. In this case any reference type can be used as type argument to replace the unbounded type parameter in an instantiation of a generic type.
  - Alternatively can have one or several bounds. In this case the type argument that replaces the bounded type parameter in an instantiation of a generic type must be a subtype of all bounds.
- The reason for imposing a type bound on a type parameter is that the code used in the **implementation code** of the generic class is **assuming** that the type used is of a certain type or any of its subtypes, and/or that it implements a certain interface.

Generics: parameter type bounds

- The syntax for specification of type parameter bounds is:

```
<TypeParameter extends AClass & AnInterface1 & ... & AnInterfaceN>
```

i.e. a list of bounds consists of one class and/or several interfaces.

- This can lead to rather complex class declarations such as:

```
class Pair<A extends Comparable<A> & Cloneable ,
           B extends Comparable<B> & Cloneable >
      implements Comparable<Pair<A,B>>, Cloneable { ... }
```

- A `Pair` class taking two type parameters (`A` and `B`), where each parameter is of a certain type that implements `Comparable` with itself and implements `Cloneable`.
- The class itself implements `Comparable` with itself and implements `Cloneable`, i.e. instances of this class are `Comparable` with each other and are `Cloneable` and the same is assumed of both parts of the `Pair`.

Generics: parameter types, interesting cases

```
class PairUtil {
  public static <A extends Number, B extends Number> double add(Pair<A, B> p) {
    return p.getFirst().doubleValue() + p.getSecond().doubleValue();
  }

  public static <A, B> Pair<B, A> swap(Pair<A, B> p) {
    A first = p.getFirst();
    B second = p.getSecond();
    return new Pair<B, A>(second, first);
  }
}
```

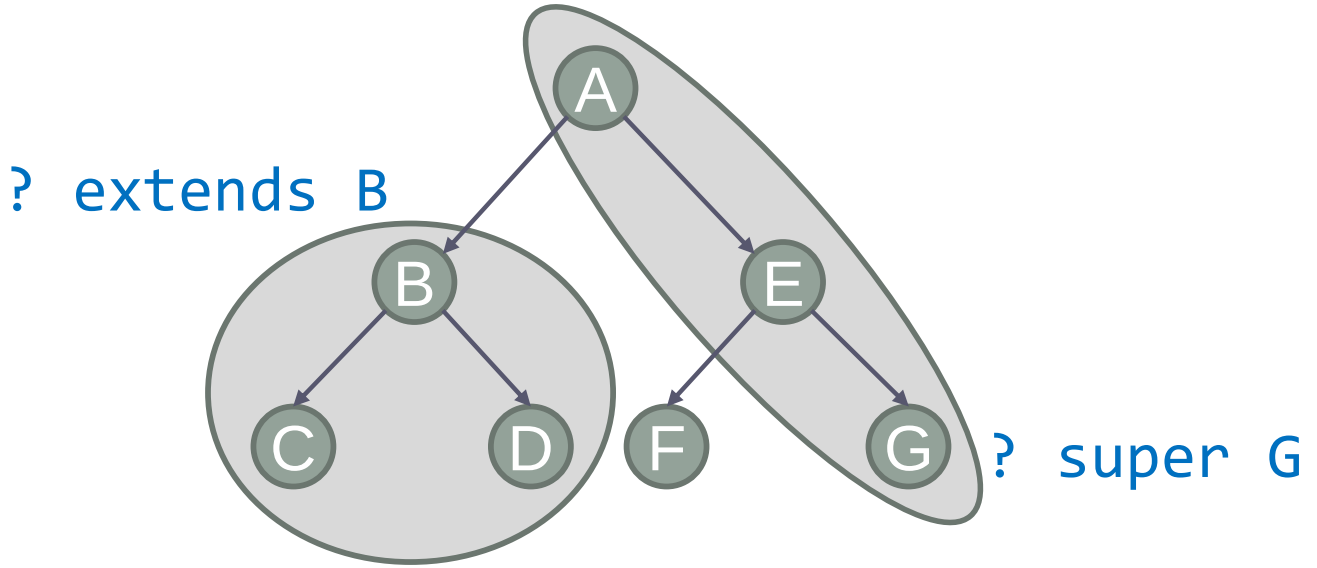
```
public class Pair <X, Y> {
  private final X a;
  private final Y b;

  public Pair(X a, Y b) {
    this.a = a;
    this.b = b;
  }
  public X getFirst() {
    return a;
  }
  public Y getSecond() {
    return b;
  }
}
```

- In the `add()` method:
  - The method is generic over two type parameters `A` and `B`, which must be subclasses of `Number`, which allows to use `doubleValue()`.
  - The argument to the method is a `Pair`; the type arguments to that `Pair` are constrained by the type parameter bounds to `add()`.
- In the `swap()` method:
  - The type parameters are used to define the return type, as well as the argument.
  - Local variables in the method are declared in terms of the type parameters.
  - The type parameters are used as type arguments in the constructor call.

Java Generics: wildcards

- A wildcard is a syntactic construct that is used to denote a family of types in a generic class/method **instantiation**. As opposed to type parameters, which are used in a generic class **definition**.
- Wildcards can be unbound or bound:
  - " `?` " : the unbounded wildcard. It stands for the family of all types.
  - " `? extends SuperType` " : a wildcard with an upper bound. It stands for the family of all types that are `SuperType` itself or subtypes of `SuperType`.
  - " `? super SubType` " - a wildcard with a lower bound. It stands for the family of all types that are `SubType` itself or supertypes of `SubType`.



Java Generics: wildcards

```
ArrayList<?> aList;
aList = new ArrayList<String>();
aList = new ArrayList<Integer>();
```

- The variable `aList` can be used to refer to any kind of `ArrayList`

```
public void printCollection( Collection<?> c ){
  for (Object o : c){
    System.out.println(o);
  }
}
```

- The method `printCollection()` receives a parameter that is a `Collection` of elements of any type. It can do so because it does not apply any type-specific operation on the `Collection` it receives as a parameter.

```
public static <T> void copy(List<? extends T> src, List<? super T> dest) {
  for (int i=0; i<src.size(); i++)
    dest.set(i,src.get(i));
  }
}
```

- The method `copy()` receives as parameters two lists, where the type of elements in the source `List` must be a subtype of type of elements in the destination `List`. Failure to impose such a restriction may allow to attempt to copy the elements of a list into a list of elements of an unrelated type, leading to an illegal type cast. It must do so because it uses `get()` and `set()`, which are bound to the type of values stored in the `List`.

Generic classes: instantiation vs. type parameter

```
Basket b = new Basket();                    // OK but using raw type!
Basket b1 = new Basket<Fruit>();            // OK but using raw type!
Basket<Fruit> b2 = new Basket<Fruit>();     // OK !

// Type mismatch: cannot convert from Basket<Fruit> to Basket<Apple>
Basket<Apple> b3 = new Basket<Fruit>();     // WRONG !!!

// Type mismatch: cannot convert from Basket<Apple> to Basket<Fruit>
Basket<Fruit> b4 = new Basket<Apple>();     // WRONG !!!

Basket<?> b5 = new Basket<Apple>();         // OK!

// 1. Cannot instantiate the type Basket<?>
// 2. Type mismatch: cannot convert from Basket<?> to Basket<Apple>
Basket<Apple> b6 = new Basket<?>();         // WRONG !!!
```

```
class Basket<E> {...}

class Fruit {...}
class Apple extends Fruit {...}
class Orange extends Fruit {...}
```

- If a generic class is referred to without using any type parameter, it refers to a **raw type**, i.e. a class that has been subject to type erasure, taking the generic type's uppermost allowable type as type argument. In the example above, a raw `Basket` is a `Basket` of `Object`.
- The class hierarchies of the parameter types are not transposed onto the generic classes that use them as parameters.
- Wildcards must be used in a context where they can be verified to instantiate to a specific type at runtime, e.g. `new Basket<?>()` is invalid as it attempts to create a `Basket` containing a value of unknown type.

Java generic class example: generic Pair class

```
/**
 *  Immutable generic pair class
 */
public class Pair<TypeOfFirst, TypeOfSecond>{
   private final TypeOfFirst first;
   private final TypeOfSecond second;

   public Pair(){}

   public Pair(TypeOfFirst first, TypeOfSecond second){
     this.first = first;
     this.second = second;
   }

   public Pair(Pair<TypeOfFirst, TypeOfSecond> newPair){
     this.first = newPair.getFirst();
     this.second = newPair.getSecond();
   }

   public TypeOfFirst getFirst() {
     return this.first;
   }

   public TypeOfSecond getSecond() {
     return this.second;
   }

   public String toString(){
   return   first.getClass().getName()  + ":" + first.toString() + " , "
          + second.getClass().getName() + ":" + second.toString();
   }

}
```

Java generic class example: generic Pair class

```
public class PairDriver {

  public static void main(String[] args) {
    Pair<String, Integer> p1 = new Pair<String, Integer>("Hello", 1);
    System.out.println(p1);

    ArrayList<Integer> v1 = new ArrayList<Integer>();
    for (int x = 1; x <= 3; x++)
      v1.add(new Integer(x));

    ArrayList<String> v2 = new ArrayList<String>();
    v2.add(new String("un"));
    v2.add(new String("deux"));
    v2.add(new String("trois"));

    ArrayList<Pair<Integer, String>> v3 = new ArrayList<Pair<Integer, String>>();
    for (int x = 0; x <= 2; x++)
      v3.add(new Pair<Integer, String>(v1.get(x), v2.get(x)));

    for (Pair<Integer, String> p : v3)
      System.out.println(p);
  }
}
```

```
java.lang.String:Hello , java.lang.Integer:1
java.lang.Integer:1 , java.lang.String:un
java.lang.Integer:2 , java.lang.String:deux
java.lang.Integer:3 , java.lang.String:trois
```

Java generic class example: generic comparable Pair class

```
public class ComparablePair<TypeOfFirst extends Comparable<TypeOfFirst>, TypeOfSecond extends Comparable<TypeOfSecond>>
  implements Comparable<ComparablePair<TypeOfFirst, TypeOfSecond>> {

  private TypeOfFirst first;
  private TypeOfSecond second;

  public ComparablePair() {}

  public ComparablePair(TypeOfFirst first, TypeOfSecond second) {
    this.first = first;
    this.second = second;
  }

  public ComparablePair(ComparablePair<TypeOfFirst, TypeOfSecond> newComparablePair) {
    this.first = newComparablePair.getFirst();
    this.second = newComparablePair.getSecond();
  }

  public String toString() {
    return first.getClass().getName() + ":" + first.toString() + " , "
           + second.getClass().getName() + ":" + second.toString();
  }

  public TypeOfFirst getFirst() { return this.first; }

  public TypeOfSecond getSecond() { return this.second; }

  public int compareTo(ComparablePair<TypeOfFirst, TypeOfSecond> otherComparablePair) {
    int compareFirst = first.compareTo(otherComparablePair.getFirst());
    int compareSecond = second.compareTo(otherComparablePair.getSecond());
    if (compareFirst != 0) {
      return compareFirst;
    } else {
      return compareSecond;
    }
  }
}
```

Java generic class example: generic comparable Pair class

```
public class ComparablePairDriver {

  public static void main(String[] args) {
    ArrayList<ComparablePair<Integer, String>> alcp = new ArrayList<ComparablePair<Integer, String>>();
    alcp.add(new ComparablePair<Integer, String>(3,"trois"));
    alcp.add(new ComparablePair<Integer, String>(4,"quatre"));
    alcp.add(new ComparablePair<Integer, String>(1,"un"));
    alcp.add(new ComparablePair<Integer, String>(1,"one"));
    alcp.add(new ComparablePair<Integer, String>(1,"one"));

    ComparablePair<Integer, String> previousalcp = null;
    for (ComparablePair<Integer, String> p : alcp) {
      System.out.println(p);
      if (previousalcp != null) System.out.println(p.compareTo(previousalcp));
      previousalcp = new ComparablePair<Integer, String>(p);
    }
  }
}
```

```
java.lang.Integer:3 , java.lang.String:trois
java.lang.Integer:4 , java.lang.String:quatre
1
java.lang.Integer:1 , java.lang.String:un
-1
java.lang.Integer:1 , java.lang.String:one
-6
java.lang.Integer:1 , java.lang.String:one
0
```

References

- Angelika Langer. Java Generics FAQ
- Gilad Bracha. Generics in the Java Programming Language.
- Paul Gibson. Generics (in Java)
- David R. Musser, Alexander A. Stepanov. *Generic Programming*. In International Symposium on Symbolic and Algebraic Computation (ISSAC 1988). Lecture Notes in Computer Science 358, Springer-Verlag, 1989, pp 13-25.
- Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. *A Comparative Study of Language Support for Generic Programming*. SIGPLAN Not. 38, 11 (October 2003), 115-134. doi=10.1145/949343.949317
- Charles W. Krueger. *Software reuse*. ACM Comput. Surv. 24, 2 (June 1992), 131-183. doi=10.1145/130844.130856
- Ross Tate, Alan Leung, Sorin Lerner. *Taming Wildcards in Java's Type System*. Technical Report. Cornell University.
- Joseph A. Goguen. *Parameterized Programming*. IEEE Trans. Softw. Eng. 10, 5 (September 1984), 528-543. doi=10.1109/TSE.1984.5010277

References

- Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. *Adding wildcards to the Java programming language*. In Proceedings of the 2004 ACM symposium on Applied computing (SAC '04). ACM, New York, NY, USA, 1289-1296. doi=10.1145/967900.968162
- java.boot.by. SCJP Tiger Study Guide. Collections/Generics.
- java2novice.com. Java Generics Sample Code.
- Oracle Corporation. The Java Tutorials. Type Erasure.