JAVA Programming

Generics

Overview

- What are Generics
- What generics do for us
- Generic classes and interfaces
- Wild cards
- Generic Methods
- Arrays of parameterized types
- Generic examples in J2SE Framework
- Under the surface
- Generics and legacy code

Generics

- What are generics
  - Generics allow a type or a method to operate on various types but remain typesafe at compiletime.
  - A generic type uses one or more *type parameters* You specify the *actual types* when you invoke it.
  - A generic method may have one or more *type parameters* for its arguments. The *actual type* is implied when you call the method.

Generics

- What generics do
  - Make your code typesafe
  - Reduce the amount of code
  - Make your code more reusable
- Beware of
  - New syntax
  - Developing a generic class requires a higher abstraction level
  - *Using* the pre-built generic types is not so difficult, building your own is more difficult.

Generics

What generics do

- Design level:
  - You can define classes, interfaces and methods with generic types.
  - You can specify the actual type when using the classes and interfaces. When calling a generic method, the type is implied.
- Code level:
  - Write type-safe code

    In Java, a program is type-safe if it compiles without errors and does not encounter ClassCastExceptions at runtime.
  - No need for casting,

    type information is passed via type arguments.

Generics

Generic classes and interfaces

- A generic interface or class takes one or more *formal type parameters* (between brackets).

```
public class Foo<type param 1, type param 2>
{
        ...use type params here ...
}
```

- A type with *formal type parameters* is called a *parameterized type.*
- A parameterized type defines a collection of different (though related) types.

Generics

Generic classes and interfaces

– Meaning of type parameter should be documented, e.g.:
  - interface List<E> type of contained objects
  - interface Map<K,V> type of keys and values
  - interface Comparable<T> type to compare with
  - class Class<T> type of represented object

– Generics enable compile-time type checking so your code is *type* safe.

Generics

Generic classes and interfaces

- Invocations
  – An invocation is a usage of a parameterized type (a declaration or instantiation).
  – Provide the *actual types* as *arguments*.
  – The *actual type* must be a *reference type*.
  – The compiler will perform type-checking wherever the type is used.

Generics

Generic classes and interfaces

Simple *instructive* example:

– Parameterized type *Data*:

```
public class Data<E>
{
    public E info;
}
```

– Invocations of *Data*:

```
Data<Integer> di = new Data<Integer>();
di.info = new Integer(10);

Data<String> ds = new Data<String>();
ds.info = "Some usefull string";
```

Generics

- Example: *Interface List<E>*

```
// List that contains (sub types of) Integer
List<Integer> li = new ArrayList<Integer>();
li.add(10);


// List that contains (sub types of) String
List<String> ls = new ArrayList<String>();
ls.add("Some string");
```

Generics

Should this be allowed:

```
List<Integer> li = new ArrayList<Integer>();
List<Object> lo = li;
```

No because this would be possible:

```
lo.add("Some string");   // li would no contain a String
```

Note the difference with arrays,
this compiles with no errors:

```
Integer[] numbers = new Integer[5];
Object[] object = numbers;
object[0] = "Some string";
```

Runtime Exception!

Generics

- So: does this compile?
  - Data<Number> dn = new Data<Number>();

  - Data<String> ds = dn;
  - Data<Integer> di = dn;
  - Data<Object> do = dn;

- Conclusion:

  The argument used for declaration and instantiation must be '*exactly*' the same. However, you may want to relax this a bit, and you can by using *wild cards*

Generics

Wild cards (?)

- To use a parameterized type with an unknown (arbitrary) type argument.
- Type is not known: you may use *Object* methods
- Type is not known: you may not assign to it.

- Example:

```
public void printList(List<?> l)
 {
   for(int i=0; i<l.size(); i++)
   {
     System.out.println(l.get(i).toString());
   }
   l.add(new Object());                          // not allowed!
 }
```

Generics

- Wild cards can be made more specific (bounded)
- Example:

  Write a method that serializes all objects in a List. The objects must be Serializable.

  Is this solution adequate:

```
public void serializeObjects(List<Serializable> list)
{
   for(Serializable obj : list)
   {
       //Serialize obj
   }
}
```

Should use a wildcard with upper bound:

```
public void serializeObjects(List<? extends Serializable> list)
...
```

Generics

- Wild cards can also have a lower bound
- Example:

  Write a method that adds a Car to any List that may contain Cars

```
public void addCar(List<? super Car> list, Car car)
 {
   list.add(car);
 }
```

```
List<Object> objects = new ArrayList<Object>();
List<Vehicle> vehicles = new ArrayList<Vehicle>();

util.addCar(objects, new Car());
util.addCar(vehicles, new Car());
```

Generics

- In summary:

  Bounded wild cards
  - Upper bound:
    - Denoted as <? extends T>
    - Actual type parameter must be a subtype of T
  - Lower bound:
    - Denoted as <? super T>
    - Actual type parameter must be a supertype of T

Generics

Generic Methods

- Methods with one or more *type parameters*.

```
private <T> void foo(T t)
{
    "use T and t here "
}
```

- *Actual* type is determined by compiler (implied) (you do not specify it in calling code).

```
foo("Test");                          // String implied
foo(new Integer(10));                 // Integer implied
```

Generics

Generic Methods

- Exercise:

  Look at the following code, define *copyInfo*:

```
Data<String> ds  = new Data<String>();
Data<Integer> di = new Data<Integer>();

copyInfo("Hello", ds);    // copies a String into Data object
copyInfo(66, di);         // copies an Integer into Data object
```

  Solution:

```
private <T> void copyInfo(T s, Data<T> d)
{
   d.info = s;
}
```

Generics

Generic Methods

- Type variables may also be bounded
  - E.g. an array of type *T* may also contain objects instantiated from derived types of *T*.

```
private <T, D extends T> void copy(D[] src, T[] dest)
{
    for(int i=0; i<src.length; i++)
    {
        dest[i] = src[i];
    }
}
```

Generics

Wild Card or *generic method* ?

Use Wild Card:

When type parameter is used only once and type itself is not required in method, you should use a wild card. So use:

```
private void print(Data<?> d)
{
    System.out.println(d.info.toString());
}
```

and not (although legal):

```
private <T> void print(Data<T> d)
{
    System.out.println(d.info.toString());
}
```

Generics

Wild Card or *generic method* ?

Use generic method:

– If parameter types are dependent:

```
private <T> void copyInfo(T s, Data<T> d)
{
   d.info = s;
}
```

– If parameter type needs to be bounded:

```
private <T, D extends T> void copy(D[] src, T[] dest)
{
    for(int i=0; i<src.length; i++)
    {
        dest[i] = src[i];
    }
}
```

Generics

Arrays of parameterized types

- Arrays contain type information of contained type

  Runtime check if inserted values are valid

```
Integer[] list = new Integer[5];
Object[] object = list;
object[0] = "Some string";                // runtime exception
```

- Parameterized types are *erased* by compiler

  Runtime the parameter type is not known

  E.g. *Data<Integer>* and *Data<String>* are both erased to *Data* (containing an *Object*)

```
Data<Integer>[] list = new Data<Integer>[3];
Object[] object = list;
object[0] = new Data<String>();   // would succeed runtime
```

Generics

Arrays of parameterized types

You can only instantiate an array of a parameterized type with an unbound wildcard as a parameter.

```
Data<?>[] arr  = new Data<?>[2];   // can contain all kinds of Data
Object[] objs = arr;
objs[0] = new Data<String>();      // you have allowed this
objs[1] = new Data<Integer>();
```

```
private void test(Data<?>[] arr )
{ code }

private <T> void test(Data<T>[] arr )
{ code  }
```

Generics

Generic examples in J2SE Framework

– Collections
– Class literal as factory object

Generics

Collections:

- Part of List interface:

```
public interface List<E> extends Collection<E>
{
  void add(int index, E element);
  Iterator<E> iterator();
}
```

- Iterator interface:

```
public interface Iterator<E>
{
  boolean hasNext();
  E next();
  void remove();
}
```

Generics

Collections:

TreeSet class (implements SortedSet).

Constructor with Comparator argument:

```
class TreeSet<E>
{
   public TreeSet(Comparator<E> comp)   { … }
}
```

But: comparator must be able to compare objects of type E *or super types of E*.

```
class TreeSet<E>
{
   public TreeSet(Comparator<? super E> comp)   { … }
}
```

Generics

Under the surface

Generics are implemented by *type erasure*

- Compiler erases all generic type information.

  Runtime only the *raw type* exists.
- Types are converted to their upper type (usually *Object*) and appropriately casted whenever the resulting code is not type-correct.

- Advantage:
  - Legacy code (with only *raw* types) and generic code can interoperate.
- Disadvantage:
  - Parameter type-information is not available at run-time.

Generics

Under the surface

- Compiler erases all generic type information.
  - Runtime type parameters do not exist.

```
List<String> lst = new ArrayList<String>();
if ( lst instanceof List<String> )          // illegal
```

  - You cannot cast to a specific parameterized type.

    Unchecked warning, may get ClassCastException

```
public <T> T castObject(Object o)
{
   return (T)o;                             // unchecked warning
}
```

```
Integer i = Util.castObject(20);            // ok
Integer j = Util.castObject("Hai");        // ClassCastException
```

Generics

Generic version of *Collections.max*:

```
public static <T> T max(Collection<T> c)
// T must implement Comparable →
```

```
public static <T extends Comparable<T>> T max(Collection<T> c)
// T must be comparable only with one of it's super types →
```

```
public static <T extends Comparable<? super T>> T max(Collection<T> c)
// after erasure method must return an Object type (old contract) →
```

```
public static <T extends Object & Comparable<? super T>>
   T max(Collection<T> c)
// Collection may also contain derived types of T   →
```

```
public static <T extends Object & Comparable<? super T>>
   T max(Collection<? extends T> c)
```

Generics

- When to use:
  - Invocate generic libraries with type parameters. Avoid using *raw* types.
  - Consider making your own libraries generic if profitable.
  - When upgrading old libraries, take care you do not change the contracts.

- Note:

  You must deploy your code on a 5.0 or more recent Virtual Machine

Generics

- Exercises

Generics

- Exercise:

  Write a method *printList* that prints out all objects contained in a List.

  Solution:

```
public static void printList(List<?> list)
{
    Iterator<?> iter = list.iterator();
    while( iter.hasNext() )
    {
        System.out.println(iter.next().toString());
    }
}
```

Generics

- Exercise:

  Write a copy method that copies all objects from one list to the other.

  Solution:

```
public static <T> void copy(List<? extends T> src, List<? super T> dest)
{
   Iterator<? extends T> iter = src.iterator();
   while( iter.hasNext() )
   {
      dest.add(iter.next());
   }
}
```

Lab: Generics