Java Generics

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Generic types

- Generics allow developers to abstract over *types*
  - Classes, interfaces, and methods can be *parameterized* by types
- The effect of using generics is *type-safe code:*
  - If the code compiles without errors or warnings, then it will not throw a typecasting exception at run-time
- Generics make code easier to read
  - Once you get used to the syntax

Use of generics

- The most common use of generics is with collections of elements
- For example, the task is to write a *Stack* class that defines standard methods for manipulating a stack of elements
  - However, the stack must be able to operate with any referential type
- Prior to Java 1.5, this task was easy to solve by simply using *Object* as the element type

Stack without generics

- Definition:

```
class Stack {
  void push(Object element) { ... }
  void pop() { ... }
  Object top() { ... }
}
```

- Use:

```
Stack stack = new Stack();
// push two integers
stack.push(new Integer(7));
stack.push(new Integer(8));
// get the top element - typecast required!
Integer n = (Integer)stack.top();
```

Stack without generics - problems

- When getting an element from the stack, a typecast is required
  - Although the programmer knows what kind of data has been placed on the stack, the *compiler* doesn't
- The “loss of type” can lead to problems if different types of elements are added:

```
stack.push(new Integer(8));
stack.push(new Float(7));
Integer k = (Integer)stack.top(); // ClassCastException
```

- The biggest problem is that this error is signalled only at run-time, during program execution

Solution: defining and using generics

- When using generics, collections are, by definition, not tied to any particular element type
- A *generic type* is defined by putting type parameter inside "<" and ">" after the class/interface name:

```
class Stack<E> {
  void push(E element) { ... }
  void pop() { ... }
  E top() { ... }
}
```

- The given class is said to be parameterized; the parameter *E* is a *type parameter*

Solution: defining and using generics

- When creating an object of a generic class, the type parameter is replaced with concrete *type argument*, such as *Integer, String*, etc., producing a *parameterized type*

```
// creating a stack of Strings
Stack<String> stack = new Stack<String>();
// add some elements
stack.push("foo");
stack.push("bar");
// getting an element doesn't require a typecast!
String elem = stack.top();
```

Glossary

- **Generic type / method**
  - A class or interface / method with one or more *type parameters* (`class Stack<E> { ... }`)
- **Parameterized type**
  - A type created from a *generic type* by providing an actual *type argument* per formal *type parameter* (`Stack<String>`)
- **Type parameter**
  - A place holder for a *type argument* (`E`)
- **Type argument**
  - A reference type used for instantiation of a generic type / method (`String`)

Advantages of generics

- In the previous example, a generic stack was instantiated using *String* as the argument for parameter *E*
- One might think of this as if the compiler had replaced all occurrences of *E* in class/interface definition with *String* (although this is not exactly the case)
- By having the type argument, the compiler can perform automatic typecasting
- More importantly, it won't allow any mixing of types:

```
Stack<String> stack = new Stack<String>();
Integer n = new Integer(10);
stack.push(n); // compile-time error!
```

- This assures there are no run-time typecasting errors

Type erasure

- Unlike C++ templates, generic type information exists only at compile time
- Once the compiler is certain the code is type-safe, generic information is removed in the resulting byte-code, i.e. it does not exist at run-time
- Consequence:

```
LinkedList<Integer> a = new LinkedList<Integer>();
LinkedList<String> b = new LinkedList<String>();
if (a.getClass() == b.getClass()) // true!
```

More on use of generics

- By convention, a type parameter should be named using a single capital letter

- The most commonly used type parameter names are:
  - E - Element (used extensively by Java Collections)
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S, U, V etc. - 2nd, 3rd, 4th types

More on use of generics

- A class/interface can have any number of type parameters:

```
interface Hash<K, V> { ... }
```

- A sub-class needs to specify a type parameter for every type parameter of all of its super-types:

```
interface Pair<M, N> { ... }

interface Triple<X, Y, Z> { ... }

class Quintet<A, B, C, D, E> implements
  Pair<A, B>, Triple<C, D, E> { ... }
```

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Sub-typing

- Because of inheritance, the following lines are allowed:

```
Object a = new Integer(10);
Object[] x = new String[100];
```

- So the following appears to be allowed as well:

```
Stack<Object> s = new Stack<Integer>();
```

- But this isn’t true!
- Although counter-intuitive at first, in generics there is no inheritance relationship between type arguments
- That is, parameterized types are *invariant*: for any two distinct types *T1* and *T2*, List<T1> is neither a subtype nor a super-type of List<T2>

No sub-typing of type arguments

- If sub-typing of type arguments was allowed, the code would not be type-safe:

```
Stack<Integer> si = new Stack<Integer>();
Stack<Object> so = si; // if this was allowed...
so.push(new Object());
Integer n = si.pop(); // run-time exception!
```

- In the given example, the compiler would be unable to guarantee that the correct type is always used – which is what the generics are for!
- So, when you think of it, if sub-typing was allowed, then generics would become useless

More on generics and inheritance

- The following code is still allowed, because it is type-safe:

```
Stack<Object> s = new Stack<Object>();
s.push(new Integer(10));       // allowed
s.push(new Float(1.0));        // allowed
Integer n = (Integer)s.top(); // requires typecast
```

- The last line requires typecast, by which the programmer takes full responsibility if an error occurs
- Inheritance between generic types themselves is also still valid:

```
List<Integer> list = new LinkedList<Integer>();
```

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Wildcards

- Consider the task of writing a method that outputs all elements of a collection (any collection)
- Without generics, this would be simple:

```
void printAll(Collection coll) {
  for (Object o : coll)
    System.out.println(o);
}
```

- With generics, one might be tempted to write:

```
void printAll(Collection<Object> coll) {
  for (Object o : coll)
    System.out.println(o);
}
```

- Which is wrong, as it only works with collections of *Objects*

Collection of unknown type

- The solution is to use the *unknown type argument*, denoted by the “?” wildcard:

```
void printAll(Collection<?> coll) {
  for (Object o : coll)
    System.out.println(o);
}
```

- The above method receives a *collection of unknown type*
- Use:

```
LinkedList<String> x = new LinkedList<String>();
printAll(x); // ok
```

Generic of unknown type - rules

- Elements of a collection of unknown type can be accessed only with the *Object* type:

```
void printAll(Collection<?> coll) {
  for (String o : coll) // compiler error
    System.out.println(o);
}
```

- More importantly, you cannot add arbitrary elements to it; since the compiler cannot confirm the correct type, the code would not be type-safe:

```
void printAll(Collection<?> coll) {
  coll.add(new Object()); // compiler error
}
```

Collection of shapes – example

- Consider the following example:

```
interface Shape {
  void draw();
}

class Circle implements Shape {
  private double x, y, radius;

  @Override
  public void draw() { ... }
}

class Rectangle implements Shape {
  private double x, y, width, height;

  @Override
  public void draw() { ... }
}
```

Collection of shapes – example (cont')

- The task is to write a method that draws a list of shapes:

```
void drawAll(List<Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

- As noted before, however, this method cannot be called for List<Circle> or List<Rectangle>
- The "regular" wildcard completely loses information about the type, so a typecast would be required:

```
void drawAll(List<?> shapes) {
  for (int i = 0; i < shapes.size(); i++) {
    Shape s = (Shape)shapes.get(i);
    s.draw();
  }
}
```

Bounded wildcard

- Solution: using a *upper-bounded* wildcard:

```
void drawAll(List<? extends Shape> shapes) {
  for (Shape s : shapes)
    s.draw();
}
```

- Language construction *<? extends Shape>* stands for *unknown type that extends class Shape (incl. Shape itself)*
- Upper-bounded wildcard sets the upper limit in the class hierarchy for the element type
- In this way, elements of the collection can be accessed with *Shape*, instead of only *Object* type: a more flexible approach
- Still, arbitrary elements can't be added to the collection!

Bounded wildcard

- Analogously, there is the *lower-bounded* wildcard:

```
void addNumbers(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++) {
    list.add(i);
  }
}
```

- Language construction *<? super Integer>* stands for *unknown type that is a supertype of Integer (incl. Integer)*
- Lower-bounded wildcard sets the lower limit in the class hierarchy for the element type
- Here, elements **can** be added to the collection

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Bounded type parameters

- Upper-bounding can also be used on type parameters when declaring a generic type, i.e. in order to limit the set of types that can be used with the generic type
- Example:

```
class StackNum<E extends Number> {
  public void push(E e) {   }
  public E top(E e) { return null; }
  public void pop() {   }
}

class StackNumbers {
  StackNum<Integer> sn = new StackNum<Integer>();
  // compiler error
  StackNum<String> ss = new StackNum<String>();
}
```

- Lower-bounding of type parameters is not allowed

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Generic methods

- In addition to classes and interfaces, methods can also be parameterized
- Type parameters for the generic method are specified in the method definition, just before the method return type
- A generic method is used to express dependencies among the types of one or more of its arguments and/or its return type
- The most common use of generic methods: modifying a generic collection

Generic method example

- Example task: write a generic method that takes an array and a list, and puts all elements from the array into the list
- Method definition:

```
private <T> void add(T[] array, List<T> list) {
  for (T elem : array)
    list.add(elem);
}
```

Generic method example

- Use:

```
String[] sa = { "foo", "bar" };
List<String> sl = new ArrayList<String>();
add(sa, sl); // T "becomes" String

Integer[] ia = { 1, 2, 3, 4 };
List<Integer> il = new ArrayList<Integer>();
add(ia, il); // T "becomes" Integer

// compiler error: T cannot be both String and
// Integer at the same time
add(sa, il);
```

Generic methods rules

- As before, the compiler is assuring the code to be type-safe, so a collection of unknown type cannot be altered even within a generic method:

```
<T> void add(T[] array, List<? extends T> list) {
  for (T elem : array)
    list.add(elem); // compiler error
}
```

- More than one type parameter can be defined as well:

```
<T, S> void add(T[] array, List<S> list) {
  // but the following cannot be allowed, as
  // T and S might be completely different types
  for (T elem : array)
    list.add(elem);
}
```

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Raw type and legacy code

- Raw type: generic type instantiated without the actual type argument:

```
LinkedList list = new LinkedList<Integer>();
```

- Technically, this should be an error, as type information is lost: the resulting code is not type-safe!
- However, the compiler only issues a warning
- This is a deliberate design decision, to allow generics to interoperate with pre-existing legacy code
- Raw types should be used only when working with legacy code; new code should always be type-safe, and rely on generics

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Limitations of generics

Cannot create arrays of parameterized types

- Because of fundamental differences between arrays and generics, such as covariance vs. invariance, it is illegal to create an array of parameterized types:

```
// if the following line was allowed...
List<String>[] strings = new ArrayList<String>[1];
// ... the code would not have been type safe:
List<Integer> ints = Arrays.asList(42);
Object[] objects = strings;
objects[0] = ints;
String s = strings[0].get(0);
```

- A workaround is possible using raw types and casting

Limitations of generics

Cannot create instances of type parameters

- Creating instances of type parameters using the **`new`** operator is forbidden:

```
static <E> void append(List<E> list) {
  E elem = new E(); // compile-time error
  list.add(elem);
}
```

- A workaround is possible using reflection

Limitations of generics

Method overload limitations

- A class cannot have two overloaded methods that will have the same signature after type erasure:

```
class Example {
  void print(Set<String> strSet) {...}
  void print(Set<Integer> intSet) {...}
}
```

Limitations of generics

Other limitations

- Generic types cannot be instantiated with primitive types (a primitive type cannot be used as a type argument)
  - Only referential types are allowed
- Classes cannot have static fields whose types are type parameters
  - Because a static field is shared by all
- Casts and **`instanceof`** cannot be used with parameterized types
  - Due to type erasure
- It is illegal to use parameterized types as exceptions
  - Directly of indirectly extending `Throwable` is prohibited

Generics

- Generic types
- Sub-typing
- Wildcards
- Bounded type parameters
- Generic methods
- Raw type and legacy code
- Limitations of generics
- Conclusion

Conclusion

- The most important benefit of using generic classes, interfaces, and methods is type-safe code
- The code is also generally easier to read, without the need for explicit typecasting
  - Albeit workarounds can make code messy
- Generics are most commonly used with collections
- There exists a strict, but logical set of rules when writing and using generics, which ensures that the resulting code is always type-safe
- However, generics have some limitations, as was discussed