Generics

Generics

- Concepts
  - Parameterized types
    - Type specified as type parameter (or type variable)
  - Type safety
    - Eliminate casting and casting errors
    - Turn runtime errors into compile time errors
  - Develop algorithms independent data type

Type Checking

- Static type checking
  - Compile time
- Dynamic type checking
  - At runtime, objects know there type
- Java uses both static and dynamic type checking
  - Casting circumvents static type checks
  - Objects and their type may be dynamic

Generics and Type Checking

- Generics extends Java's static type checking
  - Compile time only!
- Compiler `-Xlint:unchecked` option
  - Causes compiler to emit unchecked type conversion warnings
- Language guarantee
  - If an entire application is compiled under 1.5, without unchecked warnings, it is type safe

Parameterized Types

- Specializations of a “generic” type to operate on a specific type
- Parameterized types are instantiated at compile type
  - A type instance, not an object instance
  - Exist only at runtime

Type Parameters

- Same syntax used for declaration, construction, return types and parameters

```
ParamaterizedType<TypeParam> var;
new ParamaterizedType<TypeParam>();
ParamaterizedType<TypeParam> method();
void method(ParamaterizedType<TypeParam> param);
```

  - TypeParam is the type parameter
- Convention for type parameter names
  - E, elements of a collection
  - K and V, keys and values of a map
  - T, U, and V, for general types

Parameterized Types

```
...
// Method declarations
void addNames(List<String> names);
List<String> getNames();
...
// Construction
List<String> strLst = new ArrayList<String>();
...
// Usage
addNames(strLst);
...
strLst = getNames();
```

Parameterized Types as Type Parameters

- Useful for containers of containers

```
...
List<List<String>> lstLst =
                    new ArrayList<List<String>>();
...
Map<String, List<String>> namedLists =
                   new HashMap<String, List<String>>();
...
```

Type Hierarchy

- Parameterized types form a hierarchy
  - Based on base type
  - Not based on type parameter

```
...
List<Number> numList = new ArrayList<Number>();
numList = new LinkedList<Number>();
// Illegal...
//numList = new ArrayList<Float>();

int i = 10;
numList.add(i);
numList.add(Math.PI);
...
```

Erasure

- Generics are a compile time process
- Type information is available at compile time only
  - Type information is *erased*, after compilation
- Parameterized types become raw types

```
List<String> lst = ArrayList<String>();
```

Effectively becomes...

```
List lst = ArrayList();
```

Wildcard (?) Types

- ?, read as "unknown type"
- Usage is restricted
  - May be a return type
    - Type is returned as `Object`
  - Not allowed as a method parameter
- Want a reference to any type of collection
- Not synonymous with `Object`
  - `Object` exists at compile and run time

Wildcard Types

```
...
List<?> lst;
List<Integer> intLst = new ArrayList<Integer>();
List<String>strLst = new ArrayList<String>();
strLst.add("AnyString");
lst = intLst;
lst = strLst;
//Illegal...
//String s = lst.get(0);
//lst.add((Object)"AnyString”));
Object obj = lst.get(0);
for (Object o : lst) {
    ...
}
```

...

Writing Parameterized Types

- Same syntax
- By convention use a single capital letter for the type parameter

```
public class Holder<T> {
    private T value;
    public T get() {
        return value;
    }
    public void set(T value) {
        this.value = value;
    }
}
```

Extending Parameterized Types

- Declaration specifies parameter type to subclass

```
import java.util.ArrayList;


public class CustomList<E> extends ArrayList<E> {
    public CustomList() {

       super();
    }
    ...
}
```

Restricting Type Parameters

- Restrict type parameter by using `extends` to specifying base class type parameter
- Allows methods of extended type to be called

Restricting Type Parameters

```
public class NumberList<E extends Number>
    extends ArrayList<E> {
    public NumberList() {
        super();
    }
    public double value(E obj) {
        return obj.doubleValue();
    }
}
...
new NumberList<Float>();
// Illegal...
//new NumberList<String>();
...
```

Restricting Type Parameters

- Type parameter may also be restricted by using `super` to specifying a class or any class higher in the class hierarchy

```
public interface Sorter<T> {
    void sort(List<T> list);
}

public class SortingList<E> extends ArrayList<E> {
    /** Allow any sorter for E or a super type of E. */
    public void setSorter(Sorter<? super E> sorter) {
        …
   }
}
```

Generic Methods

- Express dependencies between a methods arguments and/or return types

```
public static <T> T addToList(List<T> list, T value) {
    list.add(value);
    return value;
}

public static <T, S extends T >
void merge(List<T> dest, List<S> src)
{
    for (S s : src) dest.add(s);
}
```

Type Parameter Restrictions

- Scope
  - Of a type parameter on a class is the instance members
  - Of a type parameter on a method is the body of the method
  - Static methods cannot reference type parameters of the class
  - Static fields cannot use type variables
- Restrictions
  - Primitive types are not supported
  - Cannot be the target of new operator
  - Cannot be used to overload methods

Type Parameter Restrictions

```
public class Holder<T> {
    /* Illegal...
    private static T value;
    public static void method(T v) {
        ...
    }
    */
    ...
}
```

Array Restrictions

- Arrays of the type variable type are allowed
- Arrays are allowed as a type parameter
- Arrays of parameterized types are not allowed
  - Except, for unbounded wildcard types

Array Restrictions

```
public class ArrayHolder<T> {
    private T[] value;
    public T[] get() { return value; }
    public void set(T[] value) {
        this.value = value;
    }
}

List<String[]> listArray = new ArrayList<String[]>();
List<?>[] listArray = new ArrayList<?>[10];
// Illegal...
//List<String>[] listArray = new ArrayList<String>[10];
```

Parameterized Exceptions

- Exceptions cannot be parameterized
  - Unable identify type in `catch`, parameterized type is not available at runtime

Class<T>

- The `Class` class is now generic
  - The type of `Class<String>` is `String.class`
- Allows more type safety when using reflection

Class<T>

```
public interface SpecialType {

   void initialize();
}

public class Factory<T extends SpecialType> {
    private Class<T> type;
    public Factory(Class<T> type) {
        this.type = type;
    }
    public T newInstance() throws InstantiationException,
                                  IllegalAccessException {
        T newObj = type.newInstance();  // No cast required
        newObj.initialize();
        return newObj;
    }
}
```