Motivation

Generics is important for:

- software reuse
- type safety
- optimization (fewer castings)

Important Principle:

“Everything should be as simple as possible but no simpler”

Java 5

Some features in new language

- boxing/unboxing
- new form of loop
- functions with variable number of arguments
- generics
- more concurrency features

Java 5: Example

```
List<Integer> ints = Arrays.asList(1, 2, 3);
int s = 0;
for (int n : ints) { s += n; }
assert s == 6;
```

assert from Java 1.4

generic collection

unboxing/boxing

new loop

function with variable number of arguments
**Example in Java 1.4**

```java
List<Integer> ints = Arrays.asList(new Integer(1), new Integer(2), new Integer(3));
int s = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
    int n = ((Integer)it.next()).intValue();
    s += n;
}
assert s == 6;
```

**similar code with Array in Java 1.4**

```java
int[] ints = new int[] {1, 2, 3};
int s = 0;
for (int i = 0; i < ints.size; i++) {
    s += ints[i];
}
assert s == 6;
```

**Generics by Erasure**

Java Generics is implemented by erasure:
- simplicity
- small
- eases evolution (compatibility)

List<Integer>, List<String>, List<List<String>>
erases to just List

Anomaly : array type very different from parametric type.
- new String[size]
- new ArrayList<String>()
with the latter losing info on element type.

**Boxing and Unboxing**

Unboxed types can give better performance

Boxed type may be cached for frequent values.

```java
public static int sum (List<Integer> ints) {
    int s = 0;
    for (int n : ints) {
        s += n;
    }
    return s;
}
```

```java
public static Integer sum_Integer (List<Integer> ints) {
    Integer s = 0;
    for (Integer n : ints) {
        s += n;
    }
    return s;
}
```

**Foreach Loop**

Works with iterator and is more concise.
Kept simple – cannot use remove + multiple lists.

```java
List<Integer> ints = Arrays.asList(1, 2, 3);
int s = 0;
for (int n : ints) {
    s += n;
}
assert s == 6;
```

Compiles to

```java
for (Iterator<Integer> it = ints.iterator(); it.hasNext(); ) {
    int n = it.next();
    s += n;
}
```
**Iterator/Iterable Interfaces**

Iterator supports iteration through a collection.

Iterable allows an Iterator object to be built.

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

**Methods with Varargs**

Syntactic sugar to support Varargs.

```java
public static <E> List<E> asList (E... arr) {
    List<E> list = new ArrayList<E>();
    for (E elt : arr) list.add(elt);
    return list;
}
```

```java
List<Integer> ints = asList(1, 2, 3);
List<String> words = asList("hello", "world");
```

The above is compiled to use array.

**Methods with Varargs**

Arrays can be used to accept a list of elements.

```java
public static <E> List<E> asList (E[] arr) {
    List<E> list = new ArrayList<E>();
    for (E elt : arr) list.add(elt);
    return list;
}
```

```java
List<Integer> ints = asList(new Integer[] { 1, 2, 3 });
List<String> words = asList(new String[] { "hello", "world" });
```

Packing argument for array is cumbersome.

**Outline**

- Overview
- Subtyping and Wildcard
- Comparison and Bounds
- Declaration and Erasure
- Reification and Reflection
- Collections
  - Iterator, Iterable, Collection
  - Set, Queues, List, Maps
- Design Patterns
- Other Issues
**Subtyping and Substitutions Principle**

Subtyping Principle:
A variable of a given type may be assigned a value of any subtype of that type. The same applies to arguments.

<table>
<thead>
<tr>
<th>Type</th>
<th>Subtype of Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Number</td>
</tr>
<tr>
<td>Double</td>
<td>Number</td>
</tr>
<tr>
<td>ArrayList&lt;E&gt;</td>
<td>List&lt;E&gt;</td>
</tr>
<tr>
<td>List&lt;E&gt;</td>
<td>Collection&lt;E&gt;</td>
</tr>
<tr>
<td>Collection&lt;E&gt;</td>
<td>Iterable&lt;E&gt;</td>
</tr>
</tbody>
</table>

However, it is not sound to have:

```
List<Integer> <: List<Number>
```

But arrays may be covariant:

```
Integer[] <: Number[]
```

**Covariant and Contravariant Subtyping**

**Covariant Subtyping**:

```
List<Integer> <: List<? extends Number>
```

List of elements of any type that is a subtype of Number

**Contravariant Subtyping**:

```
List<Number> <: List<? super Integer>
```

List of elements of any type that is a supertype of Number

**Get and Put Principle**:
use an extends wildcard when you only get values out of a structure, use a super wildcard when you put values into a structure. Don’t use wildcard when you both get and put.

**Example**

Copy from one list to another:

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

Getting elements:

```java
public static double sum(Collection<? extends Number> nums) {
    double s = 0.0;
    for (Number num : nums) s += num.doubleValue();
    return s;
}
```

**Example**

Putting elements:

```java
List<Object> objs = Arrays.asList(1, "two");
List<? super Integer> ints = objs;
ints.add(3);  // ok
double dbl = sum(ints);  // compile-time error
```

Two Bounds? Not legal though plausible.

```java
double sumCount(Collection<? super Integer, extends Number> coll, int n)  
// not legal Java!
```
**Arrays**

Array subtyping is covariant.

This was designed before generics.

Seems irrelevant now:
- unsound as it relies on runtime checks
- incompatible with Collection
- should perhaps deprecate over time.

One Solution: Use more of Collection rather than Array
- more flexibility
- more features/operations
- better generics

---

**Wildcard Capture**

We can reverse a list using parametric type or wildcard type?

```java
public static void reverse(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));
    }
}
```

```java
public static void reverse(List<? extends Object> list) {
    List<Object> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1)); // compile-time error
    }
}
```

---

**Wildcard vs Type Parameter**

Wildcards may be used if only Objects are being read.

*Collection<?> also stands for Collection<? extends Object>*

```java
interface Collection<E> {
    ...
    public boolean contains (Object o);
    public boolean containsAll (Collection<?> o);
    ...
}
```

Alternative (more restrictive but safer).

```java
interface MyCollection<E> { // alternative design
    ...
    public boolean contains (E o);
    public boolean containsAll (Collection<? extends E> o);
    ...
}
```

---

**Wildcard Capture**

Solution is to use a wrapper function with wildcard capture:

```java
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));
    }
}
```

This solution is similar to a open/close capture of an existential type.
**Restriction on Wildcards**

Wildcards should not appear at
(i) top-level of class instance creation
(ii) explicit type parameters of generic method
(iii) in supertypes of extends/implements

```java
List<? generic> list = new ArrayList<>()(); // compile-time error
Map<String, ? generic> map
    = new HashMap<String, ? extends Number>(); // compile-time error

List<?> list = new ArrayList<Object>()(); // ok
List<?> list = new List<Object>()(); // compile-time error
List<?> list = new ArrayList<Object>()(); // compile-time error
```

**Restriction on Wildcards**

Restriction on supertype of extends/implements

```java
class AnyList extends ArrayList<?> (...) // compile-time error
And so is this.
class AnotherList implements List<?> (...) // compile-time error
But, as before, nested wildcards are permitted.
class NestedList extends ArrayList<List<? super Number>> [...] // ok
```

Restriction on explicit parameter of methods

```java
List<?> list = Lists.<?>factory(); // illegal
List<List<? super Number>> = Lists.<List<? super Number>>factory(); // permitted
```

**Outline**

- Overview
- Subtyping and Wildcard
- Comparison and Bounds
- Declaration and Erasure
- Reification and Reflection
- Collections
  - Iterator, Iterable, Collection
  - Set, Queues, List, Maps
- Design Patterns
- Other Issues

**Comparison and Bounds**

`x.compareTo(y)` method is based on natural ordering
- `x.less_than(y)` returns a negative value
- `x.equal_to(y)` returns zero
- `x.more_than(y)` returns a positive value

Consistency with equal
- `x.equals(y)` if and only if `x.compareTo(y)==0`

Main difference with `null`
- `x.equals(null)` returns `false`
- `x.compareTo(null)` throws an exception
**Contract for Comparable**

Anti-symmetric:

\[ \text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x)) \]

Transitivity:

\[
\begin{align*}
\text{if } x.\text{compareTo}(y) &< 0 \text{ and } y.\text{compareTo}(z) < 0 \\
\text{then } x.\text{compareTo}(z) &< 0
\end{align*}
\]

Congruence:

\[
\begin{align*}
\text{if } x.\text{compareTo}(y) = 0 \\
\text{then } \forall z. \text{sgn}(x.\text{compareTo}(z)) = \text{sgn}(x.\text{compareTo}(z))
\end{align*}
\]

**Maximum of a Collection**

Generic code to find maximum:

```java
public static <T extends Comparable<T>> T max (Collection<T> coll) {
    T candidate = coll.iterator().next();
    for (T elt : coll) {
        if (candidate.compareTo(elt) < 0) candidate = elt;
    }
    return candidate;
}
```

A more general signature is based on get/put principle:

```java
<T extends Comparable<? super T>> T max (Collection<? extends T> coll)
```

**Implementing Integer as Comparable**

Correct way:

```java
class Integer implements Comparable<Integer> {
    ...
    public int compare (Integer that) {
        return this.value < that.value ? -1 : 0:
        this.value == that.value ? 0 : 1;
    }
    ...
}
```

Incorrect way - overflow problem:

```java
class Integer implements Comparable<Integer> {
    ...
    public int compareTo (Integer that) {
        // bad implementation -- don't do it this way!
        return this.value - that.value;
    }
    ...
}
```

**Fruity Example**

There is some control over what can be compared.

```java
class Fruit {...}
class Apple extends Fruit implements Comparable<Apple> {...}
class Orange extends Fruit implements Comparable<Orange> {...}
```

**cannot compare apple with orange**

```java
class Fruit implements Comparable<Fruit> {...}
class Apple extends Fruit {...}
class Orange extends Fruit {...}
```

**can now compare between orange/apple**
### Fruity Example

Recall:

```java
<T extends Comparable<? super T>> T max (Collection<? extends T> coll)
```

This works for `List<Orange>` and `List<Fruit>`, but old version works for only `List<Fruit>`.

Orange extends Comparable<? super Orange>

And this is true because both of the following hold.

Orange extends Comparable<Fruit> and Fruit super Orange

### Comparator

Allows additional ad-hoc ordering to be specified:

```java
interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

Example: shorter string is smaller

```java
Comparator<String> sizeOrder
    = new Comparator<String> () {
        public int compare(String s1, String s2) {
            return s1.length() < s2.length() ? -1 :
                    s1.length() > s2.length() ? 1 :
                    s1.compareTo(s2) ;
    }
};
```

Implement `max` using `Comparator`:

```java
public static <T> T max (Collection<T> coll, Comparator<T> cmp) {
    T candidate = coll.iterator().next();
    for (T elt : coll) {
        if (cmp.compare(candidate, elt) < 0) { candidate = elt; }
    }
    return candidate;
}
```

Comparator from natural order:

```java
public static <T extends Comparable<? super T>>
    Comparator<T> naturalOrder () {
        return new Comparator<T>() {
            public int compare (T o1, T o2) { return o1.compareTo(o2); }
        };
    }
```

### Enumerated Types

Enumerated type corresponds to a class with a set of final static values. First, an abstract class:

```java
public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
    private final String name;
    private final int ordinal;
    protected Enum(String name, int ordinal) {
        this.name = name; this.ordinal = ordinal;
    }
    public final String name() { return name; }
    public final int ordinal() { return ordinal; }
    public String toString() { return name; }
    public final int compareTo(E o) {
        return ordinal - o.ordinal;
    }
}
```
**Enumerated Type**

An instance of enumerated type.

```java
// corresponds to
// enum Season { WINTER, SPRING, SUMMER, FALL }
final class Season extends Enum<Season> {
    private Season(String name, int ordinal) { super(name,ordinal); }
    public static final Season WINTER = new Season("WINTER",0);
    public static final Season SPRING = new Season("SPRING",1);
    public static final Season SUMMER = new Season("SUMMER",2);
    public static final Season FALL = new Season("FALL",3);
    private static final Season[] VALUES = { WINTER, SPRING, SUMMER, FALL };
    public static Season[] values() { return VALUES; }
    public static Season valueOf(String name) { for (T e : VALUES) if (e.name().equals(name)) return e;
        throw new IllegalArgumentException();
    }
}
```

**Covariant Overriding**

Java 5 can override another if arguments match exactly but the result of overriding method is a subtype of other method.

Useful for clone method:

```java
class Object {
    public Object clone() { ... }
}
class Point {
    public Point clone() { return new Point(x,y);}  
}
```

**Constructors**

Actual type parameters should be provided:

```java
Pair<String,Integer> p = new Pair<String,Integer>("one",2)
```

If you forget, it is a raw type with unchecked warning:

```java
Pair<String,Integer> p = new Pair("one",2)
```

**Outline**

- Overview
- Subtyping and Wildcard
- Comparison and Bounds
- Declaration and Erasure
- Reification and Reflection
- Collections
  - Iterator, Iterable, Collection
  - Set, Queues, List, Maps
- Design Patterns
- Other Issues
**Static Members**

Static methods are independent of any type parameters:

```java
Cell.getCount()    // ok
Cell<Integer>.getCount() // compile-time error
Cell<?>>.getCount()  // compile-time error
```

---

**Outline**

- Overview
- Subtyping and Wildcard
- Comparison and Bounds
- Declaration and Erasure
- Reification and Reflection
- Collections
  - Iterator, Iterable, Collection
  - Set, Queues, List, Maps
- Design Patterns
- Other Issues

---

**How Erasure Works**

- The erasure of List<Integer>, List<String>, and List<List<String>> is List.
- The erasure of List<Integer>[] is List[].
- The erasure of List is itself, similarly for any raw type.
- The erasure of int is itself, similarly for any primitive type.
- The erasure of Integer is itself, similarly for any type without type parameters.
- The erasure of T in the definition of asList (see Section 1.4) is Object, because T has no bound.
- The erasure of T in the definition of max (see Section 3.2) is Comparable, because T has bound Comparables<? super T>.
- The erasure of T in the later definition of max (see Section 3.6) is Object, because T has bound Object & Comparable<? super T> so we take the erasure of the leftmost bound.
- The erasure of LinkedListCollection<E>.Node or LinkedListCollection.Node<E> (see Section 3.9) is LinkedListCollection.Node.

---

**Reification**

Refers to an ability to get run-time type information. This is a kind of concretization.

Array is reified with its component type, but parameterized types is reified without its component type.

```java
Number[] has reified type Number[]
ArrayList<Number> has reified type ArrayList
```
**Reified Types**

Type that is reifiable.
- a primitive type (such as `int`).
- a non-parameterized class or interface type (such as `Number`, `String`, or `Runnable`).
- a parameterized type instantiated with unbounded wildcards (such as `List<?>, ArrayList<?>, or `Map<?, ?>`).
- a raw type (such as `List`, `ArrayList`, or `Map`).
- or an array whose component type is reifiable (such as `int[]`, `Number[]`, `List<?>[]`, `List[]`, or `int[][]`).

Type that is *not* reifiable.
- a type variable (such as `T`).
- a parameterized type with actual parameters (such as `List<Number>, ArrayList<String>`, or `Map<String, Integer>`).
- or a parameterized type with a bound (such as `List<? extends Number>` or `Comparable<? super String>`).

**Reification - Arrays**

An incorrect code to convert a collection to an array.

```java
import java.util.*;

class Annoying {
    public static <T> T[] toArray(Collection<T> c) {
        T[] a = (T[]) new Object[c.size()]; // unchecked cast
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
}
```

*not reifiable*

```java
import java.util.*;

class AlsoAnnoying {
    public static List<Integer>[] twoLists() {
        List<Integer> a = Arrays.asList(1, 2, 3);
        List<Integer> b = Arrays.asList(4, 5, 6);
        return new List<Integer>[] {a, b}; // compile-time error
    }
}
```

```
import java.util.*;

class Wrong {
    public static <T> T[] toArray(Collection<T> c) {
        T[] a = (T[]) new Object[c.size()]; // unchecked cast
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
}
```

```
public static void main(String[] args) {
    List<String> l = Arrays.asList("one", "two");
    System.out.println(l);
    String[] a = toArray(l); // class cast error
}
```

**Reification - Arrays**

More problem:

```java
import java.util.*;

class Wrong {
    public static Object[] toArray(Collection c) {
        Object[] a = (Object[]) new Object[c.size()]; // unchecked cast
        int i=0; for (Object x : c) a[i++] = x;
        return a;
    }
}
```

```
public static void main(String[] args) {
    List l = Arrays.asList(args);
    String[] a = (String[])toArray(l); // class cast error
}
```
Reification - Arrays

Alternative using another array + reflection!

```java
import java.util.*;
class Right<T> {
    public static <T> T[] toArray(Collection<T> c, T[] a) {
        if (a.length < c.size())
            a = (T[])java.lang.reflect.Array.newInstance(a.getClass().getComponentType(), c.size());
        int i=0; for (T x : c) a[i++] = x;
        if (i < a.length) a[i] = null;
        return a;
    }
    public static void main(String[] args) {
        List<String> l = Arrays.asList("one", "two");
        String[] a = toArray(l, new String[0]);
        assert Arrays.toString(a).equals("[one, two]");
        String[] b = new String[] { "x", "x", "x", "x" };
        toArray(l, b);
        assert Arrays.toString(b).equals("[one, two, null, x]");
    }
}
```

Reification

Reification is a term to allow a program to examine its own definition.

Generics for reflection supports the process using new generic programming techniques.

Reflection for generics allow generic types (e.g. type vars, wildcard types) to be captured at runtime.

Generics for Reflection

Solution using a Class – runtime type!

```java
import java.util.*;
class RightWithClass<T> {
    public static <T> T[] toArray(Collection<T> c, Class<T> k) {
        T[] a = (T[])java.lang.reflect.Array.newInstance(k, c.size());
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
    public static void main(String[] args) {
        List<String> l = Arrays.asList("one", "two");
        String[] a = toArray(l, String.class);
        assert Arrays.toString(a).equals("[one, two]");
    }
}
```

Generics for Reflection

A new generic type for Class

```java
class Class<T> {
    public T newInstance();
    public T cast(Object o);
    public Class<? super T> getSuperclass();
    public <D> Class<? extends D> asSubclass(Class<? extends D> k);
    public <A extends Annotation> A getAnnotation(Class<A> k);
    public boolean isAnnotationPresent(Class<? extends Annotation> k);
    ...
}
```
Reflection for Primitive Type

We cannot use `Class<int>` as type parameter must be reference type. Use `Class<Integer>` for `int.class` instead!

Java.lang.reflect.array.newInstance(int.class, size) returns `int[]` and not `Integer[]` through a hack!

However, `int[].class` is correctly denoted by `Class<int[]>`!

Reflection for Generic

Non-generic reflection example:

```java
public static void toString(Class<?> k) {
    System.out.println(k + " (toString)");
    for (Field f : k.getDeclaredFields())
        System.out.println(f.toString());
    for (Constructor c : k.getDeclaredConstructors())
        System.out.println(c.toString());
    for (Method m : k.getDeclaredMethods())
        System.out.println(m.toString());
    System.out.println();
}
```

Output:

```
class Cell (toString)
private java.lang.Object Cell.value
public Cell(java.lang.Object) public java.lang.Object Cell.getValue()
public static Cell Cell.copy(Cell)
public void Cell.setValue(java.lang.Object)
```

Generic Reflection Library

class GenericReflection {
    public static <T> T newInstance(T object) {
        return (T) object.getClass().newInstance(); // unchecked cast
    }
    public static <T> Class<T> getComponentType(T[] a) {
        return (Class<T>) a.getClass().getComponentType(); // unchecked cast
    }
    public static <T> T[] newArray(Class<T> k, int size) {
        if (k.isPrimitive())
            throw new IllegalArgumentException("Argument cannot be primitive: "+k);
        return (T[]) java.lang.reflect.Array.newInstance(k, size); // unchecked cast
    }
    public static <T> T[] newArray(T[] a, int size) {
        return newInstance(getComponentType(a), size);
    }
}

Reflection for Generic

Generic reflection example:

```java
public static void toGenericString(Class<?> k) {
    System.out.println(k + " (toGenericString)");
    for (Field f : k.getDeclaredFields())
        System.out.println(f.toGenericString());
    for (Constructor c : k.getDeclaredConstructors())
        System.out.println(c.toGenericString());
    for (Method m : k.getDeclaredMethods())
        System.out.println(m.toGenericString());
    System.out.println();
}
```

Output:

```
class Cell (toGenericString)
private T Cell.value
public T Cell.getValue()
public static <T> Cell<T> Cell.copy(Cell<T>)
public void Cell.setValue(T)
```

Bytecode contains generic type information!
Reflecting Generic Types

Type interface to describe generic type:

- class `Class`, representing a primitive type or raw type;
- interface `ParameterizedType`, representing a generic class or interface, from which you can extract an array of the actual parameter types;
- interface `TypeVariable`, representing a type variable, from which you can extract the bounds on the type variable;
- interface `GenericArrayType`, representing an array, from which you can extract the array component type;
- interface `WildcardType`, representing a wildcard, from which you can extract a lower or upper bound on the wildcard.