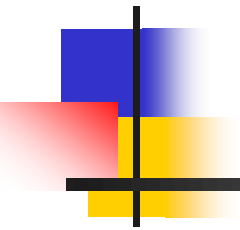
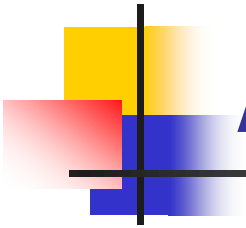


Effective Java, Chapter 5

Generics



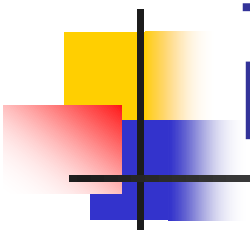
Agenda

- Material From Joshua Bloch
  - Effective Java: Programming Language Guide
- Cover Items 26-33
  - "Generics" Chapter
- Bottom Line:
  - Generics are safer, than raw types
  - But generics are also more complex
  - Raw types are allowed for backwards compatibility

Terms

Term	中文	Example	Item
Parameterized type	参数化的类型	List<String>	
Actual type parameter	实际类型参数	String	
Generic Type	泛型	List<E>	
Formal type parameter	形式化类型参数	E	
Unbounded wildcard type	无限制通配符类型	List<?>	
Raw type	原生态类型	List	
Bounded type parameter	有限制类型参数	<E extends Number>	
Recursive type bound	递归类型参数	<T extends Comparable <T>>	
Bounded wildcard type	有限制通配符类型	List<? extends Number>	
Generic method	泛型方法	static <E> List<E> asList (E[] a)	
Type token	类型令牌	String.class	

Item 26: Don't Use Raw Types in New Code



- A class (interface) with one or more type parameters is a *generic* class (interface)
- Examples:
  - `List` is a *raw* type
  - `List<E>` is a *generic* interface
  - `List<String>` is a parameterized type
    - `String` is the actual type parameter corresponding to `E`

Example: Replacing raw types

```
// Now a raw collection type – don't do this
   private final Collection stamps = …; // Contains only Stamps
// Erroneous insertion of coin into stamp collection
   stamps.add(new Coin(…));   // Oops!  We're set up for ClassCastException later

// Parameterized collection type - typesafe
     private final Collection<Stamp> stamps = …;
     stamps.add(new Coin(…));  // result is instead a compile time error, which is good
```

```
// Now a raw iterator type – don't do this!
   for (Iterator I = stamps.iterator(); i.hasNext(); ) {
      Stamp s = (Stamp) i.next();       // Throws ClassCastException
       …//  Do something with the stamp
    }
// for-each loop over parameterized collection – typesafe
   for (Stamp s: stamps) {   // No (explicit) cast
       …// Do something with the stamp
}
```

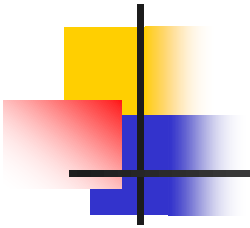
Example: List vs. List<Object>

```
// Uses raw type (List) – fails at runtime
   public static void main(String[] args) {
      List<String> strings = new ArrayList<String>();
      unsafeAdd(strings, new Integer(42));
      String s = strings.get(0);  //Exception from compiler generated cast
   }
   // note use of raw types
   private static void unsafeAdd(List list, Object o) {
      list.add(o);
   }
// There is a compile time warning:
Test.java:10: warning: unchecked call to add(E) in raw type List
   list.add(o);
            ^
// If we ignore the warning, and run the program, we get a ClassCastException
//  where the compiler inserted the cast
// If we try the following, it won't compile (see Item 25)
   private static void unsafeAdd( List<Object> list, Object o) { list.add(o);}
```

Example: Set vs. Set<?>

```
// Use of raw type for unknown element type – don't do this!
   static int numElementsInCommonSet (Set s1, Set s2) {
      int result = 0;
      for (Object o1: s1)
         { if (s2.contains(o1)) result ++; }
      return result;
   }

// Unbounded wildcard type – typesafe and flexible
   static int numElementsInCommonSet (Set<?> s1, Set<?> s2) {
      int result = 0;
      for (Object o1: s1)
         { if (s2.contains(o1)) result ++; }
      return result;
   }
// Do the question marks really buy you anything?
// Answer: Wildcard is typesafe,
// because you can't add *anything* (except null) to Collection<?>
```



Two Exceptional Cases

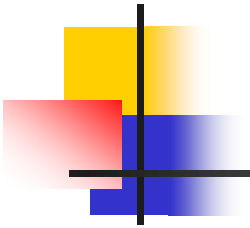
// Two exceptions: Raw types ok in

1) Class Literals: List.class, not List<String>.class
2) instanceof operator

```
if (o instanceof Set) {   // raw type ok
  Set<?> m = (Set<?>) o;  // Wildcard type
```

// Why the exceptions? Compatibility with old Java

Item 27: Eliminate Unchecked Warnings



- Generics result in many compiler warnings
  - Eliminate them
- As a last resort, suppress the warnings
  - Use the `@SuppressWarnings` annotation
  - Do so as at local level as possible
  - Never user SuppressWarnings on an entire class.
- Some are easy:

```
Set<Lark> exaltation = new HashSet();            // warning
Set<Lark> exaltation = new HashSet <Lark>();  // no warning
```

Example: Suppressing Warnings

toArray is a method of *ArrayList*; *elements* is field of *ArrayList*, which declared as "*Object[]*"

```
public <T> T[] toArray (T[] a) {
    if (a.length < size)
       return (T[]) Arrays.copyOf(elements, size, a.getClass());
    System.arraycopy(elements, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}
```

- The compiler generates a warning:

```
ArrayList.java:305: warning [unchecked] unchecked cast
found   : Object[], required T[]
    return (T[]) Arrays.copyOf(elements, size, a.getClass());
```

Example: Suppressing Warnings

- Suppressing the warning(method 1):

```
@SuppressWarnings("unchecked")
public <T> T[] toArray (T[] a) {
    if (a.length < size)
       return (T[]) Arrays.copyOf(elements, size, a.getClass());
        …
}
```

- Suppressing the warning(method 2): preferred

```
if (a.length < size) {
    // This cast is correct because the array we're creating
    // is of the same type as the one passed in, which is T[]
    @SuppressWarnings("unchecked")
     T[] result = (T[]) Arrays.copyOf(elements, size, a.getClass());
     return result; }
```

Item 28: Prefer Lists to Arrays

- Lists play well with generics
  - Generic array creation not typesafe (hence illegal)
  - No new List<E>[], new List<String>[] , or new E[]
- Arrays are *Covariant*, Generics are *Invariant*
  - If Sub is a subtype of Super
    - Then Sub[] is a subtype of Super[]
    - But List<Sub> is **not** a subtype of List<Super>
- Arrays are reified; Generics are erased
  - Generics are compile time only

Example: Covariance vs. Invariance

```
// Fails at runtime
Object[] objectArray = new Long[1];
objectArray[0] = "I don't fit in!";          // Throws ArrayStoreException

// Won't compile
List<Object> o1 = new ArrayList<Long>();
o1.add("I don't fit in!");                   //  Incompatible types
```

- Not compiling is better than a runtime exception.
- This is basically an argument for why invariance is preferable to covariance for generics.

Item 29: Favor Generic Types

- Parameterize collection declarations
  - Use the generic types
- Implementer has to work harder
  - But clients have type safety
- Stack example: How to support this?

```
public static void main (String[] args) {
    Stack<String> stack = new Stack<String>();
    for (String arg: args) { stack.push(arg);}
    while (!stack.isEmpty()) { …stack.pop()…}
}
```

Example: Converting collection to generics (without generic)

```
public class Stack {                    // Original Version – no generics
   private Object [] elements;
   private int size = 0;
   private static final int CAP = 16;

   public Stack() { elements = new Object [CAP];}

   public void push( Object e ) {
      ensureCapacity();
      elements [size++] = e;
   }

   public Object pop() {
      if (size == 0) { throw new ISE(…); }
      Object result = elements [--size];
      elements[size] = null;
      return result;
   }

   // remainder of Stack omitted – See Bloch
```

Example: Converting collection to generics (one approach)

```
public class Stack <E> {                    // First cut at generics
  private E [] elements;
  private int size = 0;
  private static final int CAP = 16;

  public Stack() {
     elements  = new  E [CAP];
  }

 public void push( E e ) {
    ensureCapacity();
    elements [size++] = e;
  }

  public E pop() {
    if (size == 0) {
       throw new ISE(…);
    }
    E result = elements [--size];
    elements[size] = null;
    return result;
  }
```

```
// error; generic array creation

  @SuppressWarning("unchecked")
    public Stack() {
       elements = new (E[]) Object [CAP];
}   // warning suppressed
```

Example: Converting collection to generics (Alternative)

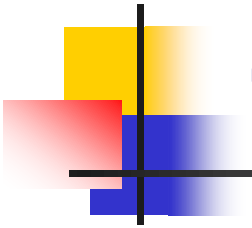
```
public class Stack <E> {
  private Object[] elements;
  private int size = 0;
  private static final int CAP = 16;

  public Stack() {
     elements  = new  Object [CAP];}

 public void push( E e ) {
    ensureCapacity();
    elements [size++] = e;
  }

  public E pop() {
    if (size == 0) { throw new ISE(…); }

    // push requires elements to be of type E, so cast is correct
    @SuppressWarning("unchecked")
    E result = (E) elements [--size];
    elements[size] = null;
    return result;
  }
```



Summary

- Generic types are safer and easier to use.
  - require no cast
- When you design new types, make sure that they can be used without such casts.
- Generify your existing types as time permits.

Item 30: Favor Generic Methods

- Just as classes benefit from generics
  - So do methods
- Writing generic methods is similar to writing generic types

```
// Uses raw types – unacceptable! (Item 23)
public static Set union (Set s1, Set s2)  {
   Set result = new HashSet(s1);              // Generates a warning
   result.addAll(s2);                         // Generates a warning
   return result;
}

// Generic method
   public static <E> Set <E> union (Set <E> s1, Set  <E> s2)  {
   Set <E> result = new HashSet <E> (s1);
   result.addAll(s2);
   return result;
}
```

Example: Recursive Type Bound

```
// Returns the maximum value in a list – uses recursive type bound
public  static <T extends Comparable<T>>  T  max (List <T> list) {
   Iterator <T> i = list.iterator();
   T result = i.next();
   while (i.hasNext()) {
      T t = i.next();       // Note: no need for a cast
      if (t.compareTo(result) > 0)
         result = t;
   }
   return result;
}
```

- Type parameter: <T extends Comparable<T>>
  - may be read as "for every type T that can be compared to itself"

Item 31: Use bounded wildcards to increase API Flexibility

```
public class Stack <E> {          // initial class
   public Stack()
   public void push( E e )
   public E pop()
   public boolean isEmpty()
}
// then we need to add a "pushAll" method

  //  pushAll method without a wildcard type – deficient!
      public void pushAll( Iterable<E> src) {
         for (E e : src) { push(e); }
      }

  //  wildcard type for parameter that serves as an E producer
      public void pushAll( Iterable<? extends E> src) {
         for (E e : src) { push(e); }
      }

// wildcard type for parameter that serves as an E consumer
      public void popAll ( Collection<? super E> dst) {
         while (!isEmpty()) { dst.add(pop()); }
      }
```

The PECS mnemonic

```
// PECS – producer extends, consumer super
// Recall earlier example
public static <E> Set <E> union (Set <E> s1, Set  <E> s2)

// Are parameters consumers or producers? ( Producers, so, extend)

public static <E> Set <E> union (Set <? extends E> s1, Set  <? extends E> s2)

// Note that return type should still be Set<E>, not Set <? extends E>
// otherwise, clients will have to use wildcards…


Set<Integer> integers = …
Set<Double> doublse = …
Set<Number> numbers = union ( integers, doubles);  // compiler error

Set<Number> numbers = union.<Number> ( integers, doubles);  // type parameter works

// max example
public static <T extends Comparable<T>>           T max (List <T>              list )   // original
public static <T extends Comparable<? super T>> T max (List<? extends T> list)   // PECS
```

Type parameter & wildcard

// **Two possible declarations for the swap method**

```
public static <E> void swap (List<E> list, int i, int j);
public static void swap (List<?> list, int i, int j);
```

- If a type parameter appears only once in a method declaration, it can be replaced with a wild card.
- Problem with the second declaration for *swap*:

e.g.

```
public static void swap(List<?> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

Trying to compile it produces this less-than-helpful error message:

```
Swap.java:5: set(int,capture#282 of ?) in List<capture#282 of ?>
cannot be applied to (int,Object)
list.set(i, list.set(j, list.get(i)));
         ^
```

(Reason: You can't put any value except null into a List<?>)

Workaround way: use a private helper method

```
public static void swap(List<?> list, int i, int j) {
    swapHelper(list, i, j);
}

// Private helper method for wildcard capture
private static <E> void swapHelper(List<E> list, int i, int j) {
    list.set(i, list.set(j, list.get(i)));
}
```

- Summary: If you write a library that will be widely used, the proper use of wildcard types should be considered mandatory.