Generics

Why Use Generics

- In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness.
- But somehow they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.
- Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on.
- Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far from the actual cause of the problem.

Why Use Generics

- Generics add stability to your code by making more of your bugs detectable at compile time.
- In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs.
- The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Why Use Generics

Code that uses generics has many benefits over non-generic code:

- Stronger type checks at compile time.
- Elimination of casts.

```
// without generics
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0); // requires casting

// use generics
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

- By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

A Non-generic Simple Box Class

```
public class Box {
    private Object object;
    public void set(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}
```

- Since its methods accept or return an Object, you are free to pass in whatever you want.
- One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

Generic Type

- A *generic type* is a generic class or interface that is parameterized over types.
- A *generic class* is defined with the following format:

    class name<T1, T2, ..., Tn> { /* ... */ }

- The type parameter section, delimited by angle brackets (<>), follows the class name.
- It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

A Generic Version of the Box Class

A *generic type declaration* is created by changing the code "public class Box" to "public class Box<T>".

This introduces the type variable, T, that can be used anywhere inside the class.

```
/**
 * Generic version of the Box class. *
@param <T> the type of the value being boxed
*/
public class Box<T> {
    // T stands for "Type" private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

A Generic Version of the Box Class

```
/**
 * Generic version of the Box class. *
@param <T> the type of the value being boxed
*/
public class Box<T> {
    // T stands for "Type" private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
```

- All occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.
- This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions.The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

- To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

```
Box<Integer> integerBox;
```

- You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* –Integer–to the Box class..
- Like any other variable declaration, this code does not actually create a new Box object.
- It simply declares that integerBox will hold a reference to a "Box of Integer“.

Invoking and Instantiating a Generic Type

- An invocation of a generic type is generally known as a parameterized type.
- To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

```
Box<Integer> integerBox = new Box<Integer>();
```

The Diamond

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context.
- This pair of angle brackets, <>, is informally called the diamond. For example, you can create an instance of Box<Integer> with the following statement:

```
Box<Integer> integerBox = new Box<>();
```

Multiple Type Parameters

```
public interface Pair<K, V> {
   public K getKey();
   public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {
   private K key;
   private V value;
   public OrderedPair(K key, V value) {
            this.key = key;
            this.value = value;
   }
   public K getKey()    { return key; }
   public V getValue() { return value; }
}
```

Multiple Type Parameters

- The following statements create two instantiations of the OrderedPair class:

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

- The code, new OrderedPair<String, Integer>, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively.
- Due to autoboxing, it is valid to pass a String and an int to the class.

Parameterized Types

- You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>).
- For example, using the OrderedPair<K, V> example:

```
OrderedPair<String, Box<Integer>> p = new  OrderedPair<>("primes", new
Box<Integer>(...));
```

Raw Types

- A raw type is the name of a generic class or interface without any type arguments.
- For example, given the generic Box class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

- To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

```
Box<Integer> intBox = new Box<>();
```

Raw Types

- If the actual type argument is omitted, you create a raw type of Box<T>:

  ```
  Box rawBox = new Box();
  ```

- Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is not a raw type.

Raw Types

- Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0.
- When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;          // OK
```

- But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();             // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;     // warning: unchecked conversion
```

Raw Types

- You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

- The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Unchecked Error Messages

- When mixing legacy code with generic code, you may encounter warning messages similar to the following:

  Note: Example.java uses unchecked or unsafe operations.

  Note: Recompile with -Xlint:unchecked for details.

- This can happen when using an older API that operates on raw types.

```
public class WarningDemo {
  public static void main(String[] args){
     Box<Integer> bi;
     bi = createBox();
  }
  static Box createBox(){
     return new Box();
  }
}
```

Unchecked Error Messages

- The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety.
- The "unchecked" warning is disabled, by default, though the compiler gives a hint.
- To see all "unchecked" warnings, recompile with -Xlint:unchecked.

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found   : Box
required: Box<java.lang.Integer>
bi = createBox();                ^
1 warning
```

- To completely disable unchecked warnings, use the -Xlint:-unchecked flag.

Generic Methods

- Generic methods are methods that introduce their own type parameters.
- This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared.
- Static and non-static generic methods are allowed, as well as generic class constructors.
- The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type.
- For static generic methods, the type parameter section must appear before the method's return type.

Define a Generic Method

```
public class Util {
    // Generic static method
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
public class Pair<K, V> {
    private K key;       private V value;
    // Generic constructor
    public Pair(K key, V value) {
        this.key = key;          this.value = value;
    }
    // Generic methods
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Invoke a Generic Method

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.<Integer, String>compare(p1, p2);
```

- The type has been explicitly provided, as shown in bold.
- Generally, this can be left out and the compiler will infer the type that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean same = Util.compare(p1, p2);
```

- This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

Bounded Type Parameters

- There may be times when you want to restrict the types that can be used as type arguments in a parameterized type.
  - For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.
  - This is what bounded type parameters are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, which in this example is Number.
- Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Bounded Type Parameters

```
public class Box<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

Multiple Bounds

- The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

  ```
  <T extends B1 & B2 & B3>
  ```

- A type variable with multiple bounds is a subtype of all the types listed in the bound.
  - If one of the bounds is a class, it must be specified first. For example:

    ```
    Class A { /* ... */ }
    interface B { /* ... */ }
    interface C { /* ... */ }
    class D <T extends A & B & C> { /* ... */ }
    ```

  - If bound A is not specified first, you get a compile-time error:

    ```
    class D <T extends B & A & C> { /* ... */ }  // compile-time error
    ```

Generic Methods and Bounded Type Parameters

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
   int count = 0;
   for (T e : anArray)
      if (e > elem)  // compiler error
         ++count;
   return count;
}
```

- The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as short, int, double, long, float, byte, and char.
- You cannot use the > operator to compare objects.

Generic Methods and Bounded Type Parameters

- To fix the problem, use a type parameter bounded by the Comparable<T> interface:

```
public interface Comparable<T> {
        public int compareTo(T o);
}
```

- The resulting code will be:

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem) {
        int count = 0;
        for (T e : anArray)
            if (e.compareTo(elem) > 0)
                ++count;
        return count;
}
```

Generics, Inheritance, and Subtypes

- It is possible to assign an object of one type to an object of another type provided that the types are compatible.
- For example, you can assign an Integer to an Object, since Object is one of Integer's supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // OK
```

- In object-oriented terminology, this is called an "is a" relationship. Since an Integer is a kind of Object, the assignment is allowed.

Generics, Inheritance, and Subtypes

Integer is also a kind of Number, so the following code is valid as well:

```
public void someMethod(Number n) { /* ... */ }
someMethod(new Integer(10));   // OK
someMethod(new Double(10.1));   // OK
```

Generics, Inheritance, and Subtypes

- The same is also true with generics. You can perform a generic type invocation, passing Number as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with Number:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

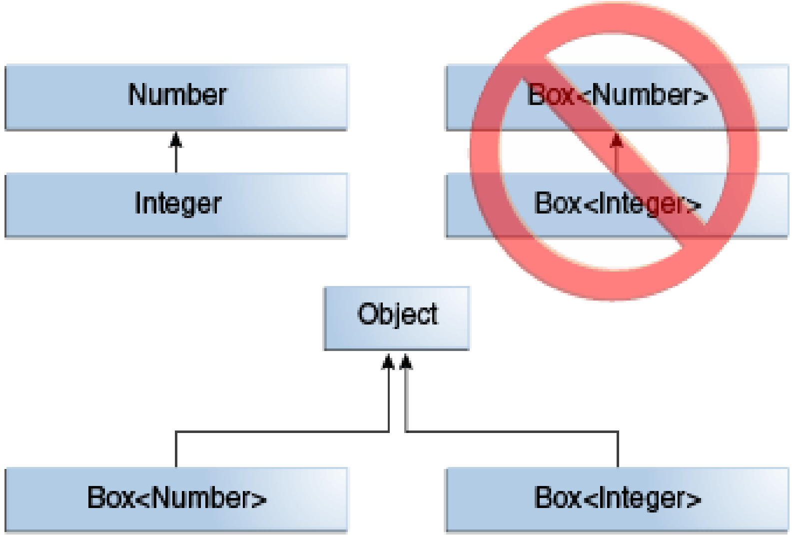
Generics, Inheritance, and Subtypes

- Now consider the following method:

  public void boxTest(Box<Number> n) { /* ... */ }

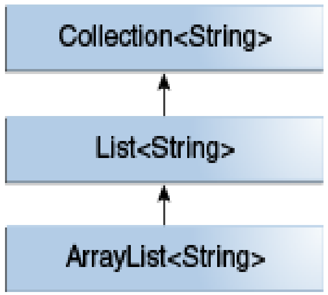
- What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>.
  - But what does that mean?
  - Are you allowed to pass in Box<Integer> or Box<Double>, as you might expect?
  - The answer is "no", because Box<Integer> and Box<Double> are not subtypes of Box<Number>.

Generics, Inheritance, and Subtypes



Generic Classes and Subtyping

- You can subtype a generic class or interface by extending or implementing it.
- The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

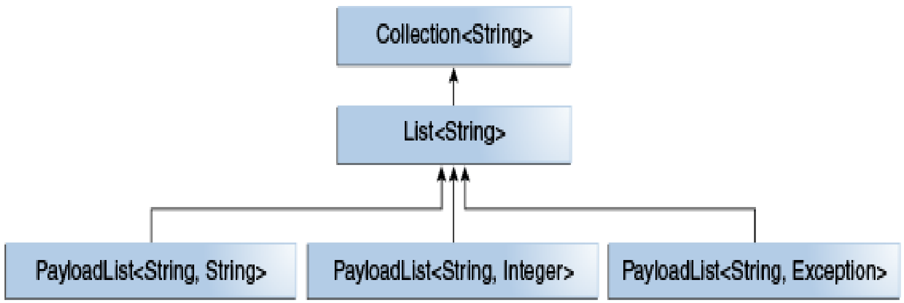


- So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

Generic Classes and Subtyping

- Now imagine we want to define our own list interface, PayloadList, that associates an optional value of generic type P with each element. Its declaration might look like:

```
interface PayloadList<E,P> extends List<E> {
    void setPayload(int index, P val);
    ...
}
```



Type Inference

- *Type inference* is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable.
- The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned.
- Finally, the inference algorithm tries to find the most specific type that works with all of the arguments.

Type Inference

- In the following example, inference determines that the second argument being passed to the pick method is of type Serializable:

```
static <T> T pick(T a1, T a2) { return a2; }
Serializable s = pick("d", new ArrayList<String>());
```

Wildcards

- In generic code, the question mark (?), called the wildcard, represents an unknown type.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific).
- The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Upper Bounded Wildcards

- You can use an upper bounded wildcard to relax the restrictions on a variable.
  - For example, say you want to write a method that works on List<Integer>, List<Double>, *and* List<Number>; you can achieve this by using an upper bounded wildcard.
- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by it s*upper bound*.
- Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

Upper Bounded Wildcards

- To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify List<? extends Number>.
- The term List<Number> is more restrictive than List<? extends Number> because the former matches a list of type Number only, whereas the latter matches a list of type Number or any of its subclasses.

Upper Bounded Wildcards

```
public static void process(List<? extends Foo> list) {
        for (Foo elem : list) {
            // ...
        }
}
```

- The upper bounded wildcard, <? extends Foo>, where Foo is any type, matches Foo and any subtype of Foo. The process method can access the list elements as type Foo.
- In the foreach clause, the elem variable iterates over each element in the list. Any method defined in the Foo class can now be used on elem.

Unbounded Wildcards

- The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a *list of unknown type*.
- There are two scenarios where an unbounded wildcard is a useful approach:
  – If you are writing a method that can be implemented using functionality provided in the Object class.
  – When the code is using methods in the generic class that don't depend on the type parameter. For example, List.sizeor List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Unbounded Wildcards

- Consider the following method, printList:

```
public static void printList(List<Object> list) {
    for (Object elem : list)
        System.out.println(elem + " ");
    System.out.println();
}
```

- The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances.
- It cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>

Unbounded Wildcards

To write a generic printList method, use List<?>:

```
public static void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
    System.out.println();
}
```

Because for any concrete type A, List<A> is a subtype of List<?>, you can use printList to print a list of any type:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<String> ls = Arrays.asList("one", "two", "three");
printList(li);
printList(ls);
```

Lower Bounded Wildcards

- A *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type.
- A lower bounded wildcard is expressed using the wildcard character (‘?’), following by the super keyword, followed by its *lower bound*: <? super A>.
- You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Lower Bounded Wildcards

- To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>.
- The term List<Integer> is more restrictive than List<? super Integer> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.

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

Wildcards and Subtyping

Given the following two regular (non-generic) classes:

```
class A { /* ... */ }
class B extends A { /* ... */ }
```

It would be reasonable to write the following code:

```
B b = new B();
A a = b;
```

This example shows that inheritance of regular classes follows this rule of subtyping: class B is a subtype of class A if B extends A.

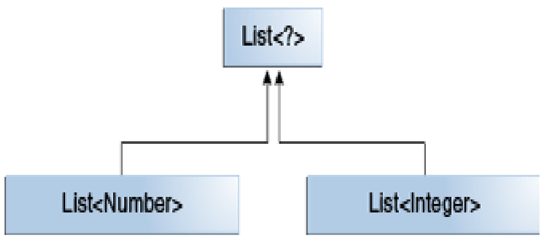
Wildcards and Subtyping

This rule does not apply to generic types:

List<B> lb = new ArrayList<>();

List<A> la = lb;   // compile-time error

Given that Integer is a subtype of Number, what is the relationship between List<Integer> and List<Number>?

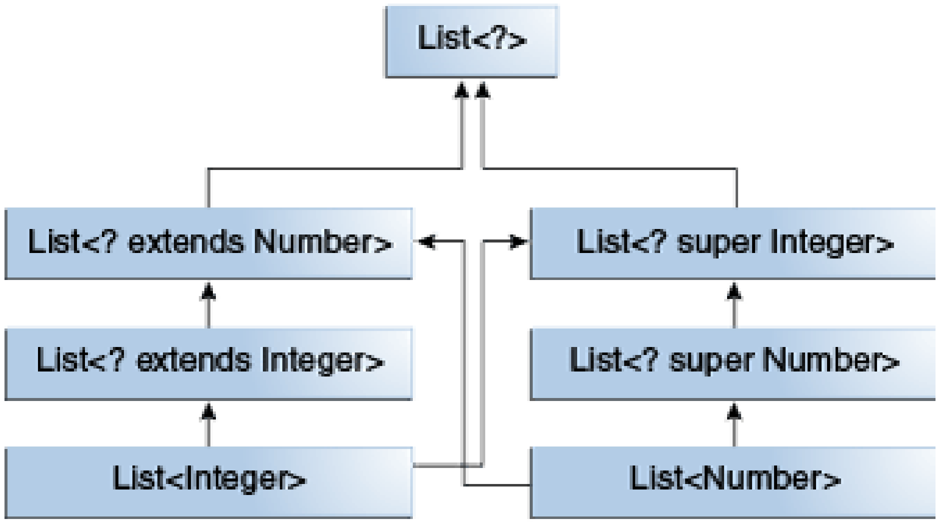


Wildcards and Subtyping

In order to create a relationship between these classes so that the code can access Number's methods through List<Integer>'s elements, use an upper bounded wildcard:

```
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number>  numList = intList;  // OK.
```

Wildcards and Subtyping



Guidelines for Wildcard Use

- **An "In" Variable**

  An “in” variable serves up data to the code. Imagine a copy method with two arguments: copy(src, dest). The src argument provides the data to be copied, so it is the "in" parameter.

- **An "Out" Variable**

  An "out" variable holds data for use elsewhere. In the copy example, copy(src, dest), the dest argument accepts data, so it is the "out" parameter.

Guidelines for Wildcard Use

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

Restrictions on Generics

Cannot Instantiate Generic Types with Primitive Types

```
class Pair<K, V> {
    private K key;
    private V value;
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    // ...
}
```

```
Pair<int, char> p = new Pair<>(8, 'a'); // compile-time error
Pair<Integer, Character> p = new Pair<>(8, 'a');
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

Cannot Create Instances of Type Parameters

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

Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
   private static T os;
   // ... }
```

Cannot Create Arrays of Parameterized Types

List<Integer>[] arrayOfLists = new List<Integer>[2]; // compile-time error

Cannot Create, Catch, or Throw Objects of Parameterized Types

```
// Extends Throwable indirectly
class MathException<T> extends Exception { /* ... */ } // compile-time error
// Extends Throwable directly
class QueueFullException<T> extends Throwable {
  /* ... */ // compile-time error
```