Generics

Bugs are simply a fact of life. Careful planning, programming, & 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 & complexity.

Some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there.
Runtime bugs, however, can be much more problematic; they don't always surface immediately, & when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

Generics add stability to code by making bugs detectable at compile time.

Extra Tutorial: Generics by Gilad Bracha.

Why Use Generics?

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.
Difference: the inputs to formal parameters are values, while the inputs to type parameters are types.

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

- Stronger type checks at compile time:

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts:

The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

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

- Enabling programmers to implement generic algorithms:

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.

Generic Types

A *generic type* is a generic class or interface that is parameterised over types.

A Simple Box Class

Box class is modified to demonstrate the concept.

Examine a non-generic Box class that operates on objects of any type.
It needs only to provide 2 methods: *set*, which adds an object to the *box*, and *get*, which retrieves it

```
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're free to pass in whatever you want, provided that it is **not** one of the **primitive types**. There is no way to verify, at compile time, how the class is used. 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.

A Generic Version of the Box Class

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.

To update the Box class to use generics, you create a generic type declaration 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.

With this change, the Box class becomes:

As you can see, 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.

```
/**
 * 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; }
}
```

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.

Most Common type parameter names

- 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

Generic 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 in this case — to the Box class itself.

Type Parameter and Type Argument Terminology: Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the `T` in `Foo<T>` is a type parameter and the `String` in `Foo<String> f` is a type argument. This lesson observes this definition when using these terms.

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", which is how Box<Integer> is read.

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.

Example: Creating an instance of Box<Integer> with the following statement: `Box<Integer> integerBox = new Box<>();`

Multiple Type Parameters

a generic class can have multiple type parameters.

EG, the generic OrderedPair class, which implements the generic Pair interface:

```
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; }
}
```

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 & 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.

As mentioned in The Diamond, because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:

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

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

You can also substitute a type parameter (that is, K or V) with a parameterized type (that is, List<String>).
Using the OrderedPair<K, V> Example:

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

Generic Types

Raw Types

A raw type is the name of a generic class or interface without any type arguments.
Eg, 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<>();
```

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 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 Boxgives 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
```

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, Avoid using raw types.

Generic 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,

eg:

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

    static Box createBox(){
        return new Box();
    }
}
```

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

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

```
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.
The @SuppressWarnings("unchecked") annotation suppresses unchecked warnings.

Unfamiliar with the @SuppressWarnings syntax? See Annotations.

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 list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

The Util class includes a generic method, compare, which compares two Pair objects:

```
public class Util {
    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;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    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; }
}
```

-Xlint:unchecked

The complete syntax for invoking this method would be:

```
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);
```

Feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, Type Inference

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, extendsis used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```
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!
    }
}
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
  be applied to (java.lang.String)
                        integerBox.inspect("10");
                                   ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

The *isEven* method invokes the *intValue* method defined in the Integer class through *n*.

Bounded Type Parameters

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 & Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array T[] that are greater than a specified element elem.

```
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. 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, & Subtypes

It's possible to assign an object of 1 type to an object of another type provided that the types are compatible. Eg, 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. But 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
```

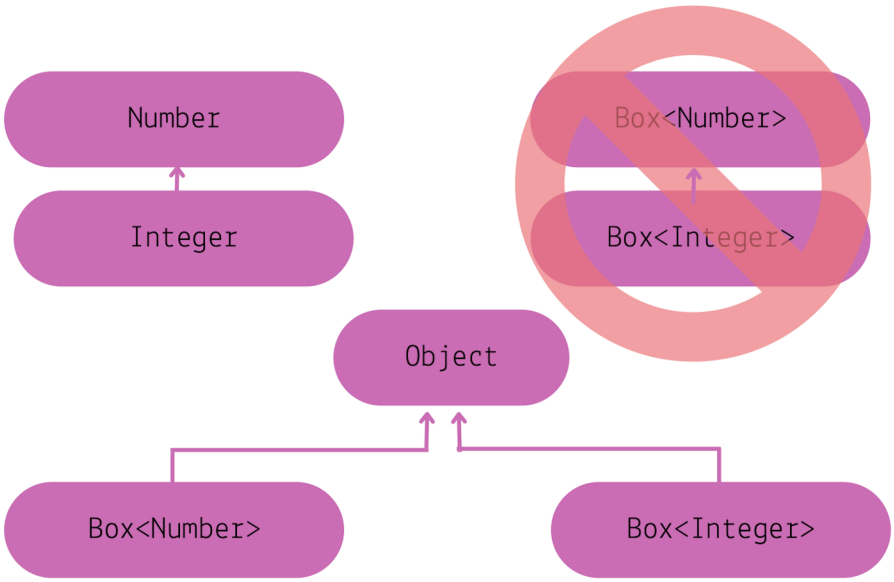
The same is also **true** with generics. You can perform a generic type invocation, passing Number as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with Number:

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

Consider the following method: `public void boxTest(Box<Number> n) { /* ... */ }`

Looking at its signature, 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>.

This is a common misunderstanding when it comes to programming with generics, but it is an NB concept to learn.



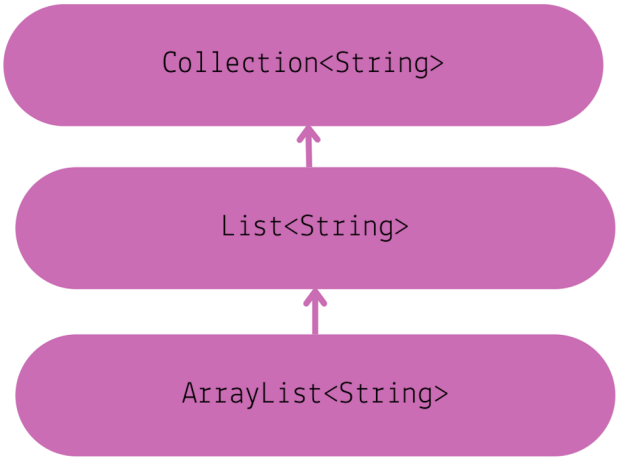
Note: Given two concrete types A and B (for example, Number and Integer), MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.

For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see Wildcards and Subtyping.

Generic Classes & 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.

Using the Collections classes as an example, ArrayList<E> implements List<E>, and List<E> extends Collection<E>. So ArrayList<String> is a subtype of List<String>, which is a subtype of Collection<String>. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.



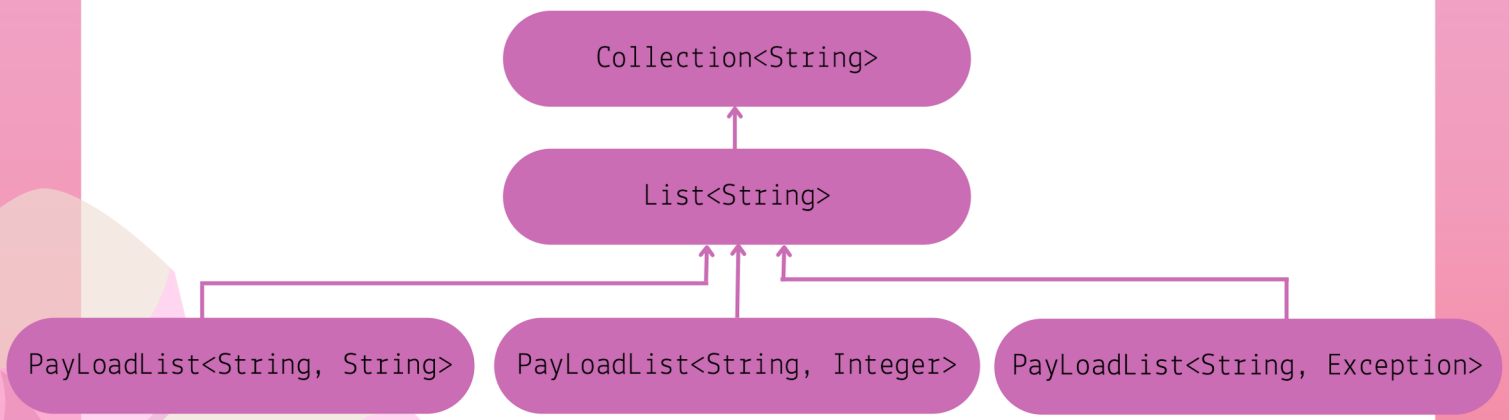
Now to define our own list interface, *PayloadList*, that associates an optional value of generic type P with each element.

Its declaration might look like:

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

The following parameterizations of PayloadList are subtypes of List<String>:

- PayloadList<String,String>
- PayloadList<String,Integer>
- PayloadList<String,Exception>



A sample *PayloadList* hierarchy

Type Inference

Type inference is a Java compiler's ability to look at each method invocation & 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 / returned. The inference algorithm tries to find the most specific type that works with all of the arguments.

Illustrating the last point above, 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>());
```

Type Inference & Generic Methods

Generic Methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets.

Example: BoxDemo, which requires the Box class

```
public class BoxDemo {

  public static <U> void addBox(U u,
      java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
    box.set(u);
    boxes.add(box);
  }

  public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
    int counter = 0;
    for (Box<U> box: boxes) {
      U boxContents = box.get();
      System.out.println("Box #" + counter + " contains [" +
          boxContents.toString() + "]");
      counter++;
    }
  }

  public static void main(String[] args) {
    java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
      new java.util.ArrayList<>();
    BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
    BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
    BoxDemo.outputBoxes(listOfIntegerBoxes);
  }
}
```

Output:

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

The generic method addBox defines one type parameter named U. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them.
Example, to invoke the generic method addBox, specify the type parameter with a type witness as follows

```
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type witness, a Java compiler automatically infers (from the method's arguments) that the type parameter is Integer:

```
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
```

Type Inference

Type Inference & Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>
();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>
();
```

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the HashMap() constructor refers to the HashMap raw type, not the Map<String, List<String>> type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Type Inference & Generic Constructors of Generic & Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

```
class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Consider the following instantiation of the class MyClass: **`new MyClass<Integer>("")`**

This statement creates an instance of the parameterized type MyClass<Integer>; the statement explicitly specifies the type Integer for the formal type parameter, X, of the generic class MyClass<X>. Note that the constructor for this generic class contains a formal type parameter, T. The compiler infers the type String for the formal type parameter, T, of the constructor of this generic class (because the actual parameter of this constructor is a Stringobject).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, compilers in Java SE 7 and later can infer the actual type parameters of the generic class being instantiated if you use the diamond (<>). Consider the following example:

```
MyClass<Integer> myObject = new MyClass<>("");
```

In this example, the compiler infers the type Integer for the formal type parameter, X, of the generic class MyClass<X>. It infers the type String for the formal type parameter, T, of the constructor of this generic class.

Note: It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

Type Inference

Target Types

The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears.

Consider the method Collections.emptyList, which is declared as follows:

```
static <T> List<T>
emptyList();
```

Consider the following assignment statement: `List<String> listOne = Collections.emptyList();`

This statement is expecting an instance of List<String>; this data type is the target type. Because the method emptyList returns a value of type List<T>, the compiler infers that the type argument T must be the value String. This works in both Java SE 7 and 8. Alternatively, you could use a type witness and specify the value of T as follows:

```
List<String> listOne = Collections.<String>emptyList();
```

However, this is not necessary in this context. It was necessary in other contexts, though. Consider the following method:

```
void processStringList(List<String> stringList)
{
    // process stringList
}
```

Suppose you want to invoke the method processStringList with an empty list. In Java SE 7, the following statement does not compile:

```
processStringList(Collections.emptyList())
;
```

The Java SE 7 compiler generates an error message similar to the following:

```
List<Object> cannot be converted to List<String>
```

The compiler requires a value for the type argument T so it starts with the value Object. Consequently, the invocation of Collections.emptyList returns a value of type List<Object>, which is incompatible with the method processStringList. Thus, in Java SE 7, you must specify the value of the value of the type argument as follows:

```
processStringList(Collections.<String>emptyList());
```

This is no longer necessary in Java SE 8. The notion of what is a target type has been expanded to include method arguments, such as the argument to the method processStringList. In this case, processStringListrequires an argument of type List<String>. The method Collections.emptyList returns a value of List<T>, so using the target type of List<String>, the compiler infers that the type argument T has a value of String. Thus, in Java SE 8, the following statement compiles:

```
processStringList(Collections.emptyList());
```

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

Use an upper bounded wildcard to relax the restrictions on a variable. Eg, writing a method that works on List<Integer>, List<Double>, and List<Number>; 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 its upper bound.
Note: in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

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.

Consider the following process method:

```
public static void process(List<? extends Foo> 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:

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

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*.

The sumOfList method returns the sum of the numbers in a list:

```
public static double sumOfList(List<? extends Number> list)
{
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

The following code, using a list of Integer objects, prints sum = 6.0:

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

A list of Double values can use the same sumOfList method. The following code prints sum = 7.0:

```
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

Wildcards

Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type.
2 scenarios where an unbounded wildcard is a **useful** approach:

1. If you are writing a method that can be implemented using functionality provided in the Object class.
2. When the code is using methods in the generic class that don't depend on the type parameter. Example, List.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

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>. 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);
```

Note: The Arrays.asList method is used in examples throughout this lesson. This static factory method converts the specified array and returns a fixed-size list.

NB:
List<Object> and List<?> are not the same. You can insert an Object, or any subtype of Object, into a List<Object>. But you can only insert null into a List<?>. The Guidelines for Wildcard Use section has more information on how to determine what kind of wildcard, if any, should be used in a given situation.

Wildcards

Lower Bounded Wildcards

The Upper Bounded Wildcards section shows that an upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword. In a similar way, a lower boundedwildcard 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>.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> — anything that can hold Integer values.
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.

The following code adds the numbers 1 through 10 to the end of a list:

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

Wildcards

Wildcards & Subtyping

Generic classes or interfaces are not related merely because there is a relationship between their types. However, you can use wildcards to create a relationship between generic classes or interfaces.

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

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

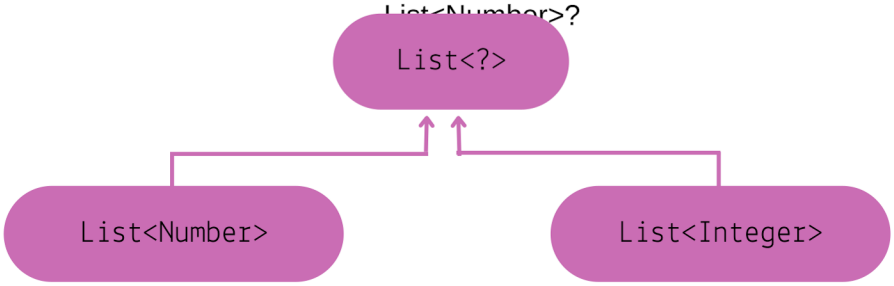
It would be reasonable to write the following code:

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

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

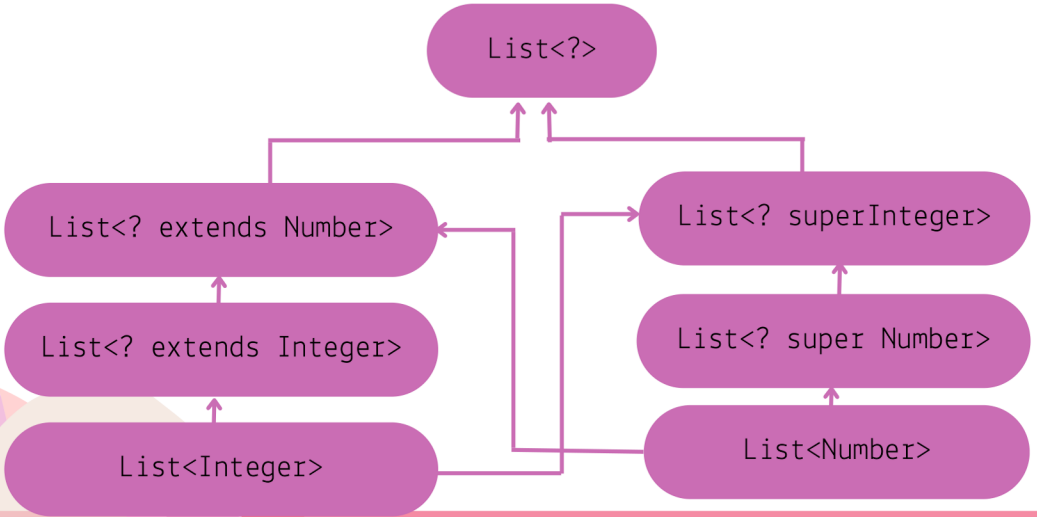


The common parent is List<?>.

Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number> and, in fact, these two types are not related. The common parent of List<Number> and List<Integer> is List<?>. 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. List<? extends Integer> is a subtype of List<? extends Number>
```

Because Integer is a subtype of Number, and numList is a list of Number objects, a relationship now exists between intList (a list of Integer objects) and numList. The following diagram shows the relationships between several List classes declared with both upper and lower bounded wildcards.



Wildcards

Wildcard Capture & Helper Methods

In some cases, the compiler infers the type of a wildcard. Example, a list may be defined as List<?> but, when evaluating an expression, the compiler infers a particular type from the code.
This scenario is known as wildcard capture.

For the most part, you don't need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The WildcardError example produces a capture error when compiled

```
import java.util.List;

public class WildcardError {

    void foo(List<?> i) {
        i.set(0, i.get(0));
    }
}
```

In this example, the compiler processes the i input parameter as being of type Object. When the foo method invokes List.set(int, E), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The WildcardError example generates the following error when compiled by Oracle's JDK 7 javac implementation:

```
WildcardError.java:6: error: method set in interface List<E> cannot be applied to given types;
    i.set(0, i.get(0));
     ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a private helper method which captures the wildcard. In this case, you can work around the problem by creating the private helper method, fooHelper, as shown in WildcardFixed:

```
public class WildcardFixed {

    void foo(List<?> i) {
        fooHelper(i);
    }


    // Helper method created so that the wildcard can be captured
    // through type inference.
    private <T> void fooHelper(List<T> l) {
        l.set(0, l.get(0));
    }

}
```

Thanks to the helper method, the compiler uses inference to determine that T is CAP#1, the capture variable, in the invocation. The example now compiles successfully.
By convention, helper methods are generally named originalMethodNameHelper.

Now consider a more complex example, WildcardErrorBad:

```
import java.util.List;

public class WildcardErrorBad {

    void swapFirst(List<? extends Number> l1, List<? extends Number> l2)
{
        Number temp = l1.get(0);
        l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
                              // got a CAP#2 extends Number;
                              // same bound, but different types
        l2.set(0, temp);      // expected a CAP#1 extends Number,
                              // got a Number
    }
}
```

Wildcards

Wildcard Capture & Helper Methods cont.

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the swapFirst method:

```
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double>  ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

While List<Integer> and List<Double> both fulfill the criteria of List<? extends Number>, it is clearly incorrect to take an item from a list of Integer values and attempt to place it into a list of Double values.

Compiling the code with Oracle's JDK javac compiler produces the following error:

```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;
      l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given
types;
      l2.set(0, temp);      // expected a CAP#1 extends Number,
        ^
  required: int,CAP#1
  found: int,Number
  reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given
types;
        i.set(0, i.get(0));
         ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
  where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
3 errors
```

There is no helper method to work around the problem, because the code is fundamentally wrong: it is clearly incorrect to take an item from a list of Integer values and attempt to place it into a list of Double values.

Wildcards

Guidelines for Wildcard Use

Wildcard Guidelines:

Determining when to use an upper bounded wildcard vs. when to use a lower bounded wildcard.

Guidelines when designing your code:

Think of variables as providing 1 of 2 functions:

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.

Some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

- 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.

These guidelines do **NOT** apply to a **method's return type**. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by List<? extends ...> can be informally thought of as read-only, but that is not a strict guarantee.

Example, the following 2 classes:

```
class NaturalNumber {

    private int i;

    public NaturalNumber(int i) { this.i = i; }
    // ...
}

class EvenNumber extends NaturalNumber {

    public EvenNumber(int i) { super(i); }
    // ...
}
```

Consider the following code:

```
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Because List<EvenNumber> is a subtype of List<? extends NaturalNumber>, you can assign le to ln. But you cannot use ln to add a natural number to a list of even numbers. The following operations on the list are possible:

- You can add null.
- You can invoke clear.
- You can get the iterator and invoke remove.
- You can capture the wildcard and write elements that you've read from the list.

You can see that the list defined by List<? extends NaturalNumber> is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time & to support generic programming.

To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Erasure of Generic Types

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

```
public class Node {

    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Object getData() { return data; }
    // ...
}
```

In the following example, the generic Node class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {

    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }
    // ...
}
```

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

```
public class Node {

    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() { return data; }
    // ...
}
```

Type Erasure

Erasure of Generic Methods

The Java compiler also erases type parameters in generic method arguments.

Consider the following generic method:

```
// Counts the number of occurrences of elem in anArray.
//
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Because T is unbounded, the Java compiler replaces it with Object:

```
public static int count(Object[] anArray, Object elem)
{
    int cnt = 0;
    for (Object e : anArray)
        if (e.equals(elem))
            ++cnt;
        return cnt;
}
```

Suppose the following classes are defined:

```
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

You can write a generic method to draw different shapes:

```
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

The Java compiler replaces T with Shape:

```
public static void draw(Shape shape) { /* ... */ }
```

Type Erasure

Effects of Type Erasure & Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. Example shows how this can occur. The following example shows how a compiler sometimes creates a synthetic method, which is called a *bridge method*, as part of the type erasure process.

Given the following two classes:

```
public class Node<T> {

    public T data;

    public Node(T data) { this.data = data; }

    public void setData(T data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node<Integer> {
    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

Consider the following code:

```
MyNode mn = new MyNode(5);
Node n = mn;            // A raw type - compiler throws an unchecked warning
n.setData("Hello");     // Causes a ClassCastException to be thrown.
Integer x = mn.data;
```

After type erasure, this code becomes:

```
MyNode mn = new MyNode(5);
Node n = mn;            // A raw type - compiler throws an unchecked warning
                        // Note: This statement could instead be the following:
                        //     Node n = (Node)mn;
                        // However, the compiler doesn't generate a cast because
                        // it isn't required.
n.setData("Hello");     // Causes a ClassCastException to be thrown.
Integer x = (Integer)mn.data;
```

The next page explains why a ClassCastException is thrown at the n.setData("Hello"); statement.

Type Erasure

Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, which is called a bridge method, as part of the type erasure process. You normally don't need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the Node & MyNode classes become:

```
public class Node {

    public Object data;

    public Node(Object data) { this.data = data; }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}

public class MyNode extends Node {

    public MyNode(Integer data) { super(data); }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

After type erasure, the method signatures do not match; the Node.setData(T) method becomes Node.setData(Object). As a result, the MyNode.setData(Integer) method does not override the Node.setData(Object)method.

To solve this problem and preserve the polymorphism of generic types after type erasure, the Java compiler generates a bridge method to ensure that subtyping works as expected.

For the MyNode class, the compiler generates the following bridge method for setData:

```
class MyNode extends Node {

    // Bridge method generated by the compiler
    //
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }

    // ...
}
```

The bridge method MyNode.setData(object) delegates to the original MyNode.setData(Integer) method. As a result, the n.setData("Hello"); statement calls the methodMyNode.setData(Object), and a ClassCastException is thrown because "Hello" can't be cast to Integer.

Type Erasure

non-Reifiable Types

The section Type Erasure discusses the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as varargs) methods whose varargs formal parameter has a non-reifiable type. See the section Arbitrary Number of Arguments in Passing Information to a Method or a Constructor for more information about varargs methods.

Topics:

- Non-Reifiable Types
- Heap Pollution
- Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters
- Preventing Warnings from Varargs Methods with Non-Reifiable Formal Parameters

---

1 Non-Reifiable Types

AA reifiable type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String> and List<Number>; the JVM cannot tell the difference between these types at runtime. As shown in Restrictions on Generics, there are certain situations where non-reifiable types cannot be used: in an instanceof expression, for example, or as an element in an array.

2 Heap Pollution

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile-time. An unchecked warning is generated if, either at compile-time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (for example, a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types, or when performing unchecked casts.

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

3 Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters

Generic methods that include vararg input parameters can cause heap pollution.

Consider the following ArrayBuilder class:

```
public class ArrayBuilder {

  public static <T> void addToList (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  public static void faultyMethod(List<String>... l) {
    Object[] objectArray = l;     // Valid
    objectArray[0] = Arrays.asList(42);
    String s = l[0].get(0);       // ClassCastException thrown here
  }

}
```

Non-Reifiable Types cont.

The following eg, HeapPollutionExample uses the ArrayBuiler class:

```
public class HeapPollutionExample {

  public static void main(String[] args) {

    List<String> stringListA = new ArrayList<String>();
    List<String> stringListB = new ArrayList<String>();

    ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
    ArrayBuilder.addToList(stringListB, "Ten", "Eleven", "Twelve");
    List<List<String>> listOfStringLists =
      new ArrayList<List<String>>();
    ArrayBuilder.addToList(listOfStringLists,
      stringListA, stringListB);

    ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
  }
}
```

When compiled, the following warning is produced by the definition of the ArrayBuilder.addToList method:

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method ArrayBuilder.addToList, the compiler translates the varargs formal parameter T... elements to the formal parameter T[] elements, an array. However, because of type erasure, the compiler converts the varargs formal parameter to Object[] elements. Consequently, there is a possibility of heap pollution.

Consequently, the compiler does not issue a warning or error if you assign a List object of any type to any array component of the objectArray array as shown by this statement:

```
Object[] objectArray = l;
```

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter l can be assigned to the variable objectArray, and thus can be assigned to l. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter List<String>... l to the formal parameter List[] l. This statement is valid; the variable l has the type List[], which is a subtype of Object[].

Consequently, the compiler does not issue a warning or error if you assign a List object of any type to any array component of the objectArray array as shown by this statement:

```
objectArray[0] = Arrays.asList(42);
```

This statement assigns to the first array component of the objectArray array with a List object that contains one object of type Integer.

Suppose you invoke ArrayBuilder.faultyMethod with the following statement:

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

At runtime, the JVM throws a ClassCastException at the following statement:

```
// ClassCastException thrown here
String s = l[0].get(0);
```

The object stored in the first array component of the variable l has the type List<Integer>, but this statement is expecting an object of type List<String>.

4 Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters

If you declare a varargs method that has parameters of a parameterized type, and you ensure that the body of the method does not throw a ClassCastException or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and non-constructor method declarations:

```
@SafeVarargs
```

The @SafeVarargs annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

```
@SuppressWarnings({"unchecked", "varargs"})
```

This approach doesn't suppress warnings generated from the method's call site. @SuppressWarnings, see Annotations.

Restrictions on Generics

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```
class Pair<K, V> {

    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // ...
}
```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V

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

You can substitute only non-primitive types for the type parameters K and V:

```
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

```
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

For more information on autoboxing, see Autoboxing and Unboxing in the Numbers and Strings lesson.

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

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

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
    E elem = cls.newInstance();   // OK
    list.add(elem);
}
```

You can invoke the append method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Restrictions on Generics

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;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field *os* is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You can't create static fields of type parameters.

Cannot Use Casts / instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
    }
}
```

The set of parameterized types passed to the *rtti* method is:

```
S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an ArrayList<Integer> and an ArrayList<String>. The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

```
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable
type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards.

Example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

Some cases the compiler knows that a type parameter is always valid and allows the cast.

Example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Restrictions on Generics

Can't Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does NOT compile:

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

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;    // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[2];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired ArrayStoreException.

Can't Create, Catch, / Throw Objects of Parameterized Types

A generic class cannot extend the Throwable class directly or indirectly.

Example, the following classes will not compile:

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

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

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
    try {
        for (J job : jobs)
            // ...
    } catch (T e) {   // compile-time error
        // ...
    }
}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T {     // OK
        // ...
    }
}
```

Can't Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

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

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

The overloads would all share the same classfile representation and will generate a compile-time error.