**Generics**

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

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.

**A Simple Box Class**

Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two 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; }

}

## 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; }

}

## Type Parameter 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

## 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");

# Raw Types

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

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

# Generic Methods

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());

}

}

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);

# 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).

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!**

}

}

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;

}

// ...

}

## Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can havemultiple 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

# Generics, Inheritance, and Subtypes

As you already know, 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. 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

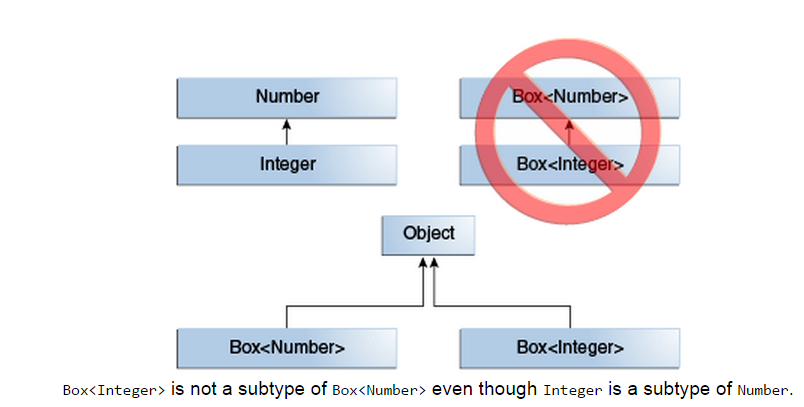
box.add(new Double(10.1)); // OK

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>.

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



Box<Integer> is not a subtype of Box<Number> even though Integer is a subtype of Number.

**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> andMyClass<B> is Object.

# Type Inference

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

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 and Instantiation of Generic Classes

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

# 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 its *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).

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.

# 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.size or List.clear. In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

# Lower Bounded Wildcards

The [Upper Bounded Wildcards](https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html) 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 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>

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);

}

}

# Guidelines for Wildcard Use

**Wildcard Guidelines:**

* 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.

## Type Erasure

## [Type Erasure](https://docs.oracle.com/javase/tutorial/java/generics/erasure.html) discusses the process where the compiler removes information related to type parameters and type arguments

## 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, 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 and 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 method becomes setData(Object) and the MyNodemethod becomes setData(Integer). Therefore, the MyNode setData method does not override the Node setData method.

To solve this problem and preserve the [polymorphism](https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html) of generic types after type erasure, a 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);

}

// ...

}

## Non-Reifiable Types

A 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.

## 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.

# Restrictions on Generics

* [Cannot Instantiate Generic Types with Primitive Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#instantiate)
* [Cannot Create Instances of Type Parameters](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createObjects)
* [Cannot Declare Static Fields Whose Types are Type Parameters](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createStatic)
* [Cannot Use Casts or instanceof With Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCast)
* [Cannot Create Arrays of Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#createArrays)
* [Cannot Create, Catch, or Throw Objects of Parameterized Types](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCatch)
* [Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotOverload)

## 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](https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html" \t "_top) in the [Numbers and Strings](https://docs.oracle.com/javase/tutorial/java/data/index.html) 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);

## 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 cannot, therefore, create static fields of type parameters.

## Cannot Use Casts or 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 anArrayList:

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. For example:

List<Integer> li = new ArrayList<>();

List<Number> ln = (List<Number>) li; // compile-time error

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

List<String> l1 = ...;

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

## Cannot 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>[]; // 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.

## Cannot Create, Catch, or Throw Objects of Parameterized Types

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

// ...

}

}

## Cannot 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 class file representation and will generate a compile-time error.