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

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.

With this change, the Box class becomes:

/\*\*

\* 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; }

}

he 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

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.

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; }

}

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");

As mentioned in [The Diamond](http://docs.oracle.com/javase/tutorial/java/generics/types.html#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");

## 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<>();

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.

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)

## Unchecked Error Messages

This can happen when using an older API that operates on raw types, as shown in the following example:

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.

# 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.

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; }

}

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);

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.

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;

}

// ...

}

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

## 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. 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

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.

## Generic Classes and Subtyping

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 ofCollection<String>. So long as you do not vary the type argument, the subtyping relationship is preserved between the types.

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);

...

}

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

* PayloadList<String,String>
* PayloadList<String,Integer>
* PayloadList<String,Exception>

# 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.

To illustrate this last point, 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>());

## Type Inference and Generic Methods

## Bounded Type Parameters:

public static <T extends Comparable<T>> T maximum(T x, T y, T z)

{

T max = x; // assume x is initially the largest

if ( y.compareTo( max ) > 0 ){

max = y; // y is the largest so far

}

if ( z.compareTo( max ) > 0 ){

max = z; // z is the largest now

}

return max; // returns the largest object

}

## Generic Classes:

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

}

public T get() {

return t;

}

public static void main(String[] args) {

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

Box<String> stringBox = new Box<String>();

integerBox.add(new Integer(10));

stringBox.add(new String("Hello World"));

System.out.printf("Integer Value :%d\n\n", integerBox.get());

System.out.printf("String Value :%s\n", stringBox.get());

}

}

# 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.

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).

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) {

// ...

}

}

public static double sumOfList(List<? extends Number> list) {

double s = 0.0;

for (Number n : list)

s += n.doubleValue();

return s;

}

# 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.

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);

# Lower Bounded Wildcards

 In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

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);

}

}

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; }

// ...

}

I'll save you the trouble...  
  
HttpSession is a session object managed by the application container. Each visitor will have a unique session object per web application they are invoking in the application container.  
  
PortletSession is a similar concept, but different implementation details. The ROOT servlet (Liferay) maintains a separate portlet session for every portlet on a page for the user that they have viewed. You view a page with 4 portlets on it, you basically have 4 portlet sessions for yourself. Portlets can access their session data, but it is separate from the portlet session of another portlet, another page, another user, etc.  
  
Just as HttpSession is a place for web apps to store their own data that they want to make available during user sessions, PortletSession is the same way. A single portlet can store it's own data to make it available during user sessions.

Now that's how it works in it's pure form. Liferay allows for some customization of how that works in order to share portlet session data, etc., but the use cases for those kinds of things are really limited. If you find yourself wanting to tinker with these settings, I think you're trying to hack something in that should really have been resolved in some other cleaner fashion.  
  
Another twist: Normally your plugins are separate wars, so there are often times when you are looking at two different HttpSessions (one managed for the ROOT web app and one for the web app that actually drives your plugin). These sessions are not the same, and something you store in one is not visible in the other.

The portletSession.APPLICATION\_SCOPE mechanism makes it possible for portlets to share session data if they are within the same portlet application. If you have two portlets that are not in the same war they are not in the same portlet application and thus, the session data is not shared either.

[PortletSession.APPLICATION\_SCOPE](http://portals.apache.org/pluto/portlet-2.0-apidocs/javax/portlet/PortletSession.html#APPLICATION_SCOPE) is designed for sharing data between different portlets that reside in the same WAR. It is basically a layer of abstraction on top of HttpSession, so the data for user1 should not be visible to user2, since each user has the same session.  
  
Do you want the users to be able to share data? If so, then you would need to keep that in application scope, and be careful to make it thread-safe.

When the user initially accesses a portlet, a PortletSession is created. The portlet  
session stores transient data associated with an individual use of the portlet.   
  
The PortletSession object extends from [HttpSession](http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html" \o "Java API" \t "_new) and serves much the same  
purpose. The PortletSession is intended to represent an ongoing conversation between the client and the portlet. To this end, the PortletSession can be used to store information needed between requests. The PortletSession is intended to store data between requests, not between portlets. As such, data stored in the session by one portlet is not accessible by another. The PortletSession is retrieved from the request object.

**Basic four lifecycle methods**

The init(PortletConfig config) is called once, immediately after a new portlet instance is created. It can be used to perform startup tasks and is akin to a servlets init method. PortletConfig represents read-only configuration data, specified in a portlet's descriptor file, portlet.xml(more on this file later). For example, PortletConfig provides access to initialization parameters.

The processAction(ActionRequest request, ActionResponse response) method is called in response to a user action such as clicking a hyperlink or submitting a form. In this method, a portlet may invoke business logic components, such as JavaBeans, to accomplish its goal. The ActionRequest and ActionResponse Interfaces are subinterfaces of PortletRequest and PortalRequest. In processAction, a portlet may modify its own state as well as persistent information about a portlet.

The render(RenderRequest request, RenderResponse response) method follows processAction in the chain of lifecycle methods. Render generates the markup that will be made accessible to the portal user. RenderRequest and RenderResponse methods, also subinterfaces of PortletRequest and PortletResponse, are available during the rendering of a portlet. The way in which the render method generates output may depend on the portlet's current state.

The destroy() method is the last lifecycle method, called just before a portlet is garbage collected and provides a last chance to free up portlet resources.