Generics

* Main purpose to bring generics is to provide type-safety to collections and to remove the type-casting while fetching from collections

Type Safety

Without generics:

List l=new ArrayList();

l.add(“String”);

l.add(10);

So without generics anything can be added in the collections. But with generics its not possible, the compiler complains if we try to do so.

List<String> l=new ArrayList<String>();

l.add(“String”);

l.add(10);

This is not possible any more. This code will not compile.

Type Casting

Without generics

List l=new ArrayList();

l.add(“String”);

String s=(String) l.get(0);

With generics we don’t have to type cast.

List<String> l=new ArrayList<String>();

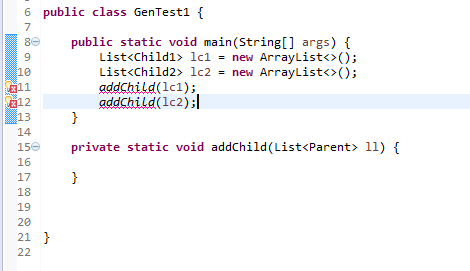
l.add(“String”);

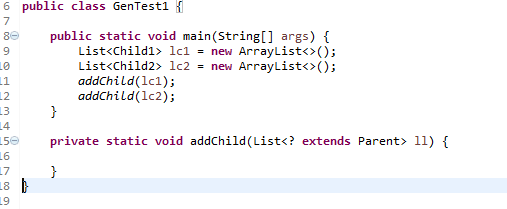
String s= l.get(0);

* The typing information does not exist at runtime! All your generic code is strictly for the compiler. Through a process called "type erasure," the compiler does all of its verifications on your generic code and then strips the type information out of the class bytecode. At runtime, ALL collection code—both legacy and new Java 5 code you write using generics—looks exactly like the pre-generic version of collections. None of your typing information exists at runtime.
* List<Parent> myList = new ArrayList<Child>(); //This doesn’t works, the type should be same.
* There IS a mechanism to tell the compiler that you can take any generic subtype of the declared argument type because you won't be putting anything in the collection. And that mechanism is the wildcard <?>.

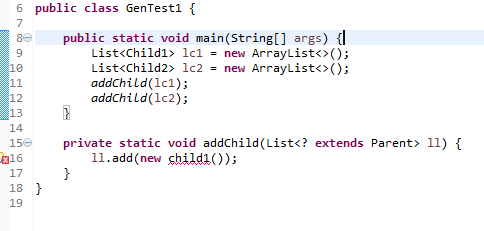
public void addAnimal(List<? extends Animal> animals)

By saying <? extends Animal>, we're saying, "I can be assigned a collection that is a subtype of List and typed for <Animal> or anything that extends Animal.





In the code snippets provided above in the first one: the method addChild has argument List<Parent>, so it will accept only list of parent type and not of child type. To solve this we change the argument to List<? extends parent>, this means that now it will accept any list that has parameter type of parent or anything that extends parent.



But we cant add anything into this list.

First, the <? extends Animal> means that you can take any subtype of Animal; however, that subtype can be EITHER a subclass of a class (abstract or concrete) OR a type that implements the interface after the word extends. In other words, the keyword extends in the context of a wildcard represents BOTH subclasses and interface implementations. There is no <? implements Serializable> syntax. If

you want to declare a method that takes anything that is of a type that implements Serializable, you'd still use extends like this: void foo(List<? extends Serializable> list)

* There is another scenario where you can use a wildcard AND still add to the collection, but in a safe way—the keyword super. Imagine, for example, that you declared the method this way:

public void addAnimal(List<? super Dog> animals) {

animals.add(new Dog()); // adding is sometimes OK with super

}

Now what you've said in this line

public void addAnimal(List<? super Dog> animals)

is essentially, "Hey compiler, please accept any List with a generic type that is of type Dog, or a supertype of Dog. Nothing lower in the inheritance tree can come in, but anything higher than Dog is OK."

* List<?>, which is the wildcard <?> without the keywords extends or super, simply means "any type." So that means any type of List can be assigned to the argument. That could be a List of <Dog>, <Integer>, <JButton>, <Socket>, whatever. And using the wildcard alone, without the keyword super (followed by a type), means that you cannot ADD anything to the list referred to as List<?>.
* List<Object> is completely different from List<?>. List<Object> means that the method can take ONLY a List<Object>. Not a List<Dog>, or a List<Cat>.
* One way to remember this is that if you see the wildcard notation (a question mark ?), this means "many possibilities". If you do NOT see the question mark, then it means the <type> in the brackets, and absolutely NOTHING ELSE.
* Which will compile and which will not

1) List<?> list = new ArrayList<Dog>();

2) List<? extends Animal> aList = new ArrayList<Dog>();

3) List<?> foo = new ArrayList<? extends Animal>();

4) List<? extends Dog> cList = new ArrayList<Integer>();

5) List<? super Dog> bList = new ArrayList<Animal>();

6) List<? super Animal> dList = new ArrayList<Dog>();

The correct answers (the statements that compile) are 1, 2, and 5.

The three that won't compile are

\* Statement: List<?> foo = new ArrayList<? extends Animal>();

Problem: you cannot use wildcard notation in the object creation. So the

new ArrayList<? extends Animal>() will not compile.

\* Statement: List<? extends Dog> cList =

new ArrayList<Integer>();

Problem: You cannot assign an Integer list to a reference that takes only a

Dog (including any subtypes of Dog, of course).

\* Statement: List<? super Animal> dList = new ArrayList<Dog>();

Problem: You cannot assign a Dog to <? super Animal>. The Dog is too "low"

in the class hierarchy. Only <Animal> or <Object> would have been legal.

* Generic Class Declaration



* Use of wildcard while declaring generic class

public class AnimalHolder<T extends Animal> { // use "T" instead

// of "?"

T animal;

public static void main(String[] args) {

AnimalHolder<Dog> dogHolder = new AnimalHolder<Dog>(); // OK

AnimalHolder<Integer> x = new AnimalHolder<Integer>(); // NO!

}

}

* Summary Points

1. Generics let you enforce compile-time type safety on Collections (or other classes and methods declared using generic type parameters).
2. When using generic collections, a cast is not needed to get (declared type) elements out of the collection. With non-generic collections, a cast is required:

List<String> gList = new ArrayList<String>();

List list = new ArrayList();

// more code

String s = gList.get(0); // no cast needed

String s = (String)list.get(0); // cast required

1. You can pass a generic collection into a method that takes a non-generic collection, but the results may be disastrous. The compiler can't stop the method from inserting the wrong type into the previously type safe collection.
2. If the compiler can recognize that non-type-safe code is potentially endangering something you originally declared as type-safe, you will get a compiler warning. For instance, if you pass a List<String> into a method declared as

void foo(List aList) { aList.add(anInteger); }

the compiler will issue a warning because the add() method is potentially an "unsafe operation."

1. Generic type information does not exist at runtime—it is for compile-time safety only. Mixing generics with legacy code can create compiled code that may throw an exception at runtime.
2. Polymorphic assignments applies only to the base type, not the generic type parameter. You can say

List<Animal> aList = new ArrayList<Animal>(); // yes

You can't say

List<Animal> aList = new ArrayList<Dog>();

1. Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

void addD(List<Dog> d) {} // can take only <Dog>

void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>

1. The wildcard keyword extends is used to mean either "extends" or "implements." So in <? extends Dog>, Dog can be a class or an interface.
2. When using a wildcard, List<? extends Dog>, the collection can be accessed but not modified.
3. When using a wildcard, List<?>, any generic type can be assigned to the reference, but for access only, no modifications.
4. List<Object> refers only to a List<Object>, while List<?> or

List<? extends Object> can hold any type of object, but for access only.

1. Declaration conventions for generics use T for type and E for element:

public interface List<E> // API declaration for List

boolean add(E o) // List.add() declaration

1. The generics type identifier can be used in class, method, and variable declarations:

class Foo<t> { } // a class

T anInstance; // an instance variable

Foo(T aRef) {} // a constructor argument

void bar(T aRef) {} // a method argument

T baz() {} // a return type

The compiler will substitute the actual type.

1. You can use more than one parameterized type in a declaration:

public class UseTwo<T, X> { }