

```
    if(local > 0) {  
        boolean positive = true; // valid only within the if block  
    } else {  
        // positive is not defined here!  
    }  
}
```

There are three basic kinds of scope for variables in Java:

- local variable, declared within a method in a class, valid for (and occupying storage only for) the time that method is executing. Every time the method is called, a new copy of the variable is used.
- instance variable, declared within a class but outside any method. It is valid for and occupies storage for as long as the corresponding object is in memory; a program can instantiate multiple objects of the class, and each one gets its own copy of all instance variables. This is the basic data structure rule of Object-Oriented programming; classes are defined to hold data specific to a "class of objects" in a given system, and each instance holds its own data.
- static variable, declared within a class as *static*, outside any method. There is only one copy of such a variable no matter how many objects are instantiated from that class.

Generics

Navigate **Language Fundamentals** topic:

Generics were added to the Java language syntax in version 1.5. This means that code using Generics will not compile with Java 1.4 and less.

Java was long criticized for the need to explicitly type-cast an element when it was taken out of a "container/collection" class. There was no way to enforce that a "collection" class contains only one type of object (e.g., to forbid *at compile time* that an Integer object is added to a Collection that should only contain Strings). This is now possible since Java 1.5.

In the first couple of years of Java evolution, Java did not have a real competitor. This has changed by the appearance of Microsoft C#. With Generics Java is better suited to compete against C#. Similar constructs to Java Generics exist in other languages, see w:Generic programming for more information.

What are Generics?

Generics are so called because this language feature allows methods to be written generically, with no foreknowledge of the type on which they will eventually be called upon to carry out their behaviors. A better name might have been **type parameter argument**. Because, it is basically that, to pass a Type as a parameter to a class at creation time.

When an object is created, parameters can be passed to the created object, through the constructor. Now with Generics, we can also pass in Types. The type-place-holders will be replaced with the specified type, before the object is created.

Type parameter arguments can be set:

for a class

When an object is created from that class the type-parameter-argument will be replaced with the actual Type.

```

public class Person<T>
{
    private Person<T> person;
    ...
}
...
// --- Create an Employee person ---
Person<Employee> empPerson = new Person<Employee>();
...
// --- Create a Customer person ---
Person<Customer> custPerson = new Person<Customer>();

```

for a method

Just like class declarations, method declarations can be generic--that is, parameterized by one or more type parameters.

```

static public <T> void assign( Person<T> person, T obj )
{
    person.setPerson( obj );
}

```

use of generics is optional

For backwards compatibility with pre-Generics code, it is okay to use generic classes without the generics type specification thing (<T>). In such a case, when you retrieve an object reference from a generic object, you will have to manually typecast it from type Object to the correct type. The compiler should also warn about unsafe operations.

Introduction

Java is a "strongly" typed language. That's why it is so easy to use. Many potential problems are caught by the compiler. One area where Java was criticized was regarding the "Container/Collection" objects. Container objects are objects that contain other objects. Before Generics were introduced there was no way to ensure that a container object contains only one type of objects. When an object was added to a container, it was automatically cast to Java Object. When it was taken out an explicit cast was needed. Normally an explicit cast is checked by the compiler.

```

String st = "This is a String";
...
Integer integer = (Integer) st; // --- Compilation Error --

```

But in the case of container classes, the compiler was not able to catch an invalid type casting.

```

1 Collection collString = new ArrayList();
2 collString.add( "This is a String" );
...
3 Integer integer = (Integer) collString.get(0); // --- No Compilation Error; RunTime CastException

```

Just looking at line 3, we do not know what type of objects *collString* contains. If that contains Integers then the code is fine.

The below code using Generic:

```
Collection<String> collString = new ArrayList<String>();
collString.add( "This is a String" );
...
Integer integer = (Integer) collString.get(0); // --- Compilation Error
```

collString is a container object, that can contain only *String* objects, nothing else, so when we get out an element it can be casted only to class that normally a *String* can be casted.

With Generics, Java strict type checking can be extended to container objects. Using Generics with container classes, gives an impression that a new container type is created, with each different type parameter. Before Generics:

```
Collection collCustomer = new ArrayList();
collCustomer.add( new Customer() );
...
Collection collObject = collCustomer; // --- No problem, both collObject and collCustomer have the same type
```

With generics:

```
Collection<Customer> collCustomer = new
ArrayList<Customer>();
collCustomer.add( new Customer() );
...
Collection<Object> collObject = collCustomer; // --- Compilation Error
```

Both *collObject* and *collCustomer* have the same type, **BUT it is against the Generic rule**, that is *collCustomer* can contain only *Customer* objects, and *collObject* can contain only *Object* object. So there is an additional check to the normal type checking, the type of the parameter type has to be matched too.

Note for C++ programmers

Java Generics are similar to C++ Templates in that both were added for the same reason. The syntax of Java Generic and C++ Template are also similar.

There are some differences however. The C++ template can be seen as a kind of macro, that generates code before compilation. The generated code depends on how the Template class is referenced. The amount of code generated depends on how many different types of classes are created from the Template. C++ Templates do not have any run-time mechanisms. The compiler creates normal code to substitute the template, similar to any 'hand-written' code.

In contrast, Java Generics are built into the language. The same Class object handles all the Generic type variations. No additional code is generated, no matter how many Generic objects are created with different type parameters. For example.

```
Collection<String> collString = new ArrayList<String>();

Collection<Integer> collInteger = new ArrayList<Integer>();
```

There is only one Class object created. In fact, at runtime, both these objects appear as the same type (both *ArrayList*'s). The generics type information is erased during compilation (type erasure). This means, for example, that if you had function that takes *Collection<T>* as an argument, and that collection happened to be empty, your function would have no way of instantiating another *T* object, because it doesn't know what *T* was.