At heart, these terms describe how the subtype relation is affected by type transformations. That is, if `A` and `B` are types, `f` is a type transformation, and ≤ the subtype relation (i.e. `A ≤ B` means that `A` is a subtype of `B`), we have

- `f` is covariant if `A ≤ B` implies that `f(A) ≤ f(B)`
- `f` is contravariant if `A ≤ B` implies that `f(B) ≤ f(A)`
- `f` is invariant if neither of the above holds

Let's consider an example. Let `f(A) = List<A>` where `List` is declared by

```
class List<T> { ... }
```

Is f covariant, contravariant, or invariant? Covariant would mean that a `List<String>` is a subtype of `List<Object>`, contravariant that a `List<Object>` is a subtype of `List<String>` and invariant that neither is a subtype of the other, i.e. `List<String>` and `List<Object>` are inconvertible types. In Java, the latter is true, we say (somewhat informally) that *generics* are invariant.

Another example. Let f(A) = `A[]`. Is f covariant, contravariant, or invariant? That is, is String[] a subtype of Object[], Object[] a subtype of String[], or is neither a subtype of the other? (Answer: In Java, arrays are covariant)

This was still rather abstract. To make it more concrete, let's look at which operations in Java are defined in terms of the subtype relation. The simplest example is assignment. The statement

```
x = y;
```

will compile only if typeof(y) ≤ typeof(x). That is, we have just learned that the statements

```
ArrayList<String> strings = new ArrayList<Object>();
ArrayList<Object> objects = new ArrayList<String>();
```

will not compile in Java, but

```
Object[] objects = new String[1];
```

will.

Another example where the subtype relation matters is a method invocation expression:

```
result = method(a);
```

Informally speaking, this statement is evaluated by assigning the value of `a` to the method's first parameter, then executing the body of the method, and then assigning the methods return value to `result`. Like the plain assignment in the last example, the "right hand side" must be a subtype of the "left hand side", i.e. this statement can only be valid if typeof(a) ≤ typeof(parameter(method)) and returntype(method) ≤ typeof(result). That is, if method is declared by:

```
Number[] method(ArrayList<Number> list) { ... }
```

none of the following expressions will compile:

```
Integer[] result = method(new ArrayList<Integer>());
Number[] result = method(new ArrayList<Integer>());
Object[] result = method(new ArrayList<Object>());
```

but

```
Number[] result = method(new ArrayList<Number>());
Object[] result = method(new ArrayList<Number>());
```

will.

Another example where subtyping matters is overriding. Consider:

```
Super sup = new Sub();
Number n = sup.method(1);
```

where

```
class Super {
    Number method(Number n) { ... }
}

class Sub extends Super {
    @Override
    Number method(Number n);
}
```

Informally, the runtime will rewrite this to:

```
class Super {
    Number method(Number n) {
        if (this instanceof Sub) {
            return ((Sub) this).method(n);  // *
        } else {
            ...
        }
    }
}
```

For the marked line to compile, the method parameter of the overriding method must be a supertype of the method parameter of the overridden method, and the return type a subtype of the overridden method's one. Formally speaking, f(A) = parametertype(method asdeclaredin(A)) must at least be contravariant, and if f(A) = returntype(method asdeclaredin(A)) must at least be covariant.

Note the "at least" above. Those are minimum requirements any reasonable statically type safe object oriented programming language will enforce, but a programming language may elect to be more strict. In the case of Java 1.4, parameter types and method return types must be identical (except for type erasure) when overriding methods, i.e. parametertype(method asdeclaredin(A)) = parametertype(method asdeclaredin(B)) when overriding. Since Java 1.5, covariant return types are permitted when overriding, i.e. the following will compile in Java 1.5, but not in Java 1.4:

```
class Collection {
    Iterator iterator() { ... }
}

class List extends Collection {
    @Override
    ListIterator iterator() { ... }
}
```

I hope I covered everything - or rather, scratched the surface. Still I hope it will help to understand the abstract, but important concept of type variance.

class A{}

```
class B extends A{}

class Collection {

   A iterator() { return new A();}

}


class List extends Collection {

   @Override

   B iterator() { return new B();}

}
```