Generics:

List<Object> o = **new** ArrayList<>();   
o.add(1); **// Autoboxing primitive int to Integer   
int** y = (Integer)o.get(0); **// unboxing Integer to a primitive int**

* A class or interface whose declaration has one or more type parameter is a generic class or interface.
* Each generic type defines a set of **parameterized types**, which consist of the class or interface name followed by an angle-bracketed list of actual type parameters corresponding to the generic types **formal type parameters**. For example, List<String> (“read list of string”) is a parameterized type representing a list whose elements are of type String. (**String** is the **actual parameter** corresponding to the **formal type parameter** **E**)
* One of the primary motivation for generics is to specify what type of object a container holds, and to have that specification backed up by the compiler. So instead of Object, we’d like to use an unspecified type which can be decided at later time. For the benefit of using a typed / wildcard parameter instead of a raw one please sees C:\Users\Mohit\IdeaProjects\OCJP\src\generics\effective\_java\RawTypesSetIssue.java
* A generic class always requires a Type parameter declaration in the class definition, e.g. class GenericHolder**<T>.**
* Tuple: group of objects wrapped together into a single object. The recipient of the object can read elements but not put new ones in.
* You can’t use primitives as type parameters.
* Even though you can say ArrayList.class, you can’t say ArrayList<Integer>.class.
* **new ArrayList<String>().getClass()** and **new ArrayList<Integer>().getClass()** , will result

In the same output **ArrayList.**

**You cannot use type parameter with a static variable , i.e static** List<T> *lsit4*;

There is no information about generic parameter type available inside generic code. Thus, you can know things like identifier of the type parameter and the bounds of the generic type- you just can’t know the actual type parameters used to create an instance (refer to C:\Users\Mohit\IdeaProjects\OCJP\src\generics\thinking\_in\_java\mystery\_of\_erasure\).

* Compiler does not allow a raw Class to extend from a typed Class ex:   
  **class** Derived3 **extends** GenericBase<T>{} **// Generates a Compiler error**

**Erasure with BaseClass and Derived Classes**

**Consider a GenericBase that takes a type parameter.**

**class** GenericBase<T>{  
 **private** T **element**;  
 **public void** set(T arg){  
 **element** = arg;  
 }  
 **public** T get(){  
 **return element**;  
 }  
}

Now we use inheritance to extend GenericBase

**class** Derived2 **extends** GenericBase<**T**>{} // **COMPILE ERROR**

**class** Derived2 **extends** GenericBase<**?**>{} // **COMPILE ERROR  
class** Derived2 **extends** GenericBase{} // **OK to use Raw type**

**Please note that the warning of using a raw type is only produced in the d2.set(obj) line.**

**public class** ErasureAndInheritance {  
 **public static void** main(String[] args){  
 Derived2 d2 = **new** Derived2();  
 Object obj = d2.get();  
 d2.set(obj); **// WARNING Unchecked call to set(..)**  
 }  
}

**Erasure and Arrays**

**public class** ArrayMaker<T> {  
 **private** Class<T> **kind**;  
 **public** ArrayMaker(Class<T> kind){  
 **this**.**kind** = kind;  
 }

**/\* Note that Arrays.newInstance() has no information about the kind that we**

**Are passing and it just returns an Object type. This must be cast to   
 T[] which then produces a warning** \*/

T[] create(**int** size){  
 **return** (T[])Array.*newInstance*(**kind**,size);  
 }  
   
 **public static void** main(String[] args){  
 ArrayMaker<String> stringMaker = **new** ArrayMaker<String>(String.**class**);  
 String[] stringArray = stringMaker.create(9);  
 **for**(**int** i=0; i < stringArray.**length**; i++)  
 stringArray[i] = String.*valueOf*(i);  
 System.***out***.println(Arrays.*toString*(stringArray));  
 }  
}

[0, 1, 2, 3, 4, 5, 6, 7, 8]

Because of erasure, anything that requires the knowledge of the exact type at run time won’t work.

1. If (arg instanceof T) **// ERROR**
2. T var = new T(); **// ERROR**
3. T[] array = new T[size**];//ERROR**
4. T[] array = (T[]) new Object[size**]// warning**

This is a valid declaration of a generic array list:

**private** List<T> **array** = **new** ArrayList<T>();

Because erasure removes type information, the only method you can call for an unbound generic parameter are those available for Object.

Bounds lets you constrain the parameter to a subset of types, then you can call the method in that subset.

(refer to C:\Users\Mohit\IdeaProjects\OCJP\src\generics\thinking\_in\_java\bounds)

Wildcards are limited to single bounds: e.g. you can’t do this:  
List<? extends SuperHearing &SuperSmell> dogboys;   
You can only do this   
List<? extends SuperHearing> audioBoys;

The Array behavior is that it holds other objects, but because we can upcast, it’s clear that the array objects can preserve the rules about the type of object they contain. It’s as if the arrays are conscious of what they are holding.

**class** Fruit{}  
**class** Apple **extends** Fruit{}  
**class** Orange **extends** Fruit{}

Fruit[] fruit = **new** Apple[10];

fruit[0] = **new** Fruit(); // **Throws Array Store exception because Array knows that the actual type is of Apple();**

fruit[0] = **new** Orange();// **Throws Array Store exception because Array knows that the actual type is of Apple();**

List<Fruit> fruitList = new ArrayList<Apple>(); **// COMPILER ERROR , incompatible types**

List<? **extends** Fruit> flist = **new** ArrayList<Apple>();  
*// flist.add(new Apple());***// COMPILER ERROR** *// flist.add(new Fruit());***// COMPILER ERROR** *// flist.add(new Object());***// COMPILER ERROR**

Fruit f = flist.get(0); // This is ok.   
List<? **extends** Fruit> flist1 = Arrays.*asList*(**new** Apple()); // This is ok.  
Apple a = (Apple) flist1.get(0); // This is ok.

Supertype wildcards : You say that the wildcard is bounded by any base class of a particular class by specifying <? Super MyClass> or even using a type parameter <? super T> , but you cannot say <T super MyClass>.

**class** Fruit{}  
**class** Apple **extends** Fruit{}  
**class** Johathan **extends** Apple{}  
**class** Orange **extends** Fruit{}

List<? super Apple> apples;

Apples.add(new Apple()); //OK

Apples.add(new Johathan()); // OK

Apples.add(new Fruit()); // **COMPILER ERROR**

The argument apples is a list of some type that is the **base type of Apple**.   
Supertype bounds relax the constraints on what you can pass into a method.

List<? **extends** Number> foo1 = **new** ArrayList<Number>(); *// Number "extends" Number (in this context)* List<? **extends** Number> foo2 = **new** ArrayList<Integer>(); *// Integer extends Number* List<? **extends** Number> foo3 = **new** ArrayList<Double>(); *// Double extends Number  
  
 /\*While Reading \*/  
  
 /\*You can read a Number because any of the lists that could be assigned to foo3 contain a Number or a subclass of Number.\*/* Number num = foo1.get(0);  
 Object ob = foo3.get(0);  
 */\*You can't read an Integer because foo3 could be pointing at a List<Double>\*/  
 //Integer in = foo1.get(0);  
  
 /\*While writing\*/  
 /\* You can't add an Integer because foo3 could be pointing at a List<Double>.  
 You can't add a Double because foo3 could be pointing at a List<Integer>.  
 You can't add a Number because foo3 could be pointing at a List<Integer>\*/  
  
 //foo1.add(1);  
 // foo2.add(new Integer(0));* foo2.add(**null**);  
 *// foo2.add(new Object());* List<? **super** Integer> foo4 = **new** ArrayList<Integer>(); *// Integer is a "superclass" of Integer (in this context)* List<? **super** Integer> foo5 = **new** ArrayList<Number>(); *// Number is a superclass of Integer* List<? **super** Integer> foo6 = **new** ArrayList<Object>(); *// Object is a superclass of Integer  
  
 /\*While Reading\*/  
  
 /\*You aren't guaranteed an Integer because foo3 could be pointing at a List<Number> or List<Object>\*/  
 //Integer in = foo4.get(0);  
  
 /\*You aren't guaranteed a Number because foo3 could be pointing at a List<Object>.\*/  
 //Number number = foo5.get(0);  
  
 /\*The only guarantee is that you will get an instance of an Object or subclass of Object (but you don't know what subclass).\*/* Object obj = foo5.get(0);  
  
 */\*While Writing\*/  
 /\*  
 You can add an Integer because an Integer is allowed in any of above lists.  
 You can add an instance of a subclass of Integer because an instance of a subclass of Integer is allowed in any of the above lists.  
\*/* foo4.add(**new** Integer(0));  
 */\*You can't add a Double because foo3 could be pointing at an ArrayList<Integer>\*/  
 // foo4.add(new Double(0));*

Refer to C:\Users\Mohit\IdeaProjects\OCJP\src\generics\wildcards\GenericReading.java on how covariance affects reading from a generic type depending upon whether if it’s from a static method or from a generic class. Basically, if you have a generic class, the parameter is established for the class when you make an instance of that class but a static generic method adapts to each method call and returns the correct type.

Wildcards

**static** List<? **extends** Object> *list3*;  
**static void** assign1(List list){  
 *list1* = list;  
 *list2* = list;  
 *list3* = list; **// Warning : Unchecked Conversion , found list, required List<? extends Object>**}

**static void** assign3(List<? **extends** Object> list){  
 *list3* = list;  
}

*assign3*(**new** ArrayList()); // Warning: Unchecked conversion. Found ArrayList required List<? extends Object>.

The compiler treats List<?> and List<? extends Object> differently. As shown above, if you try to insert a raw object on aa list of type <?> the compiler will not complain, however the compiler will produce a warning on a list of <? Extends Object>.