**Why generics?**

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.  
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.
* Elimination of casts.  
  The following code snippet without generics requires casting:
* List list = new ArrayList();
* list.add("hello");
* String s = **(String)** list.get(0);

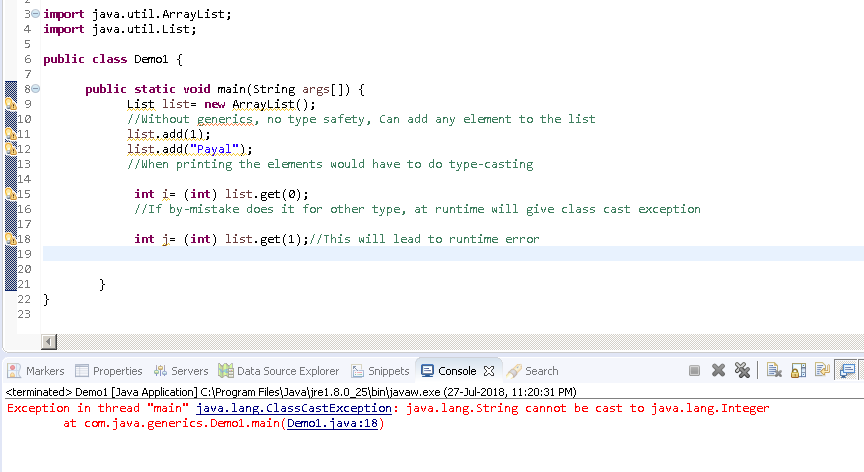
When re-written to use generics, the code does not require casting:

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

list.add("hello");

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

* Enabling programmers to implement generic algorithms.  
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.



**Generic Types**

A *generic type* is a generic class or interface that is parameterized over types.

If we create a class Box:

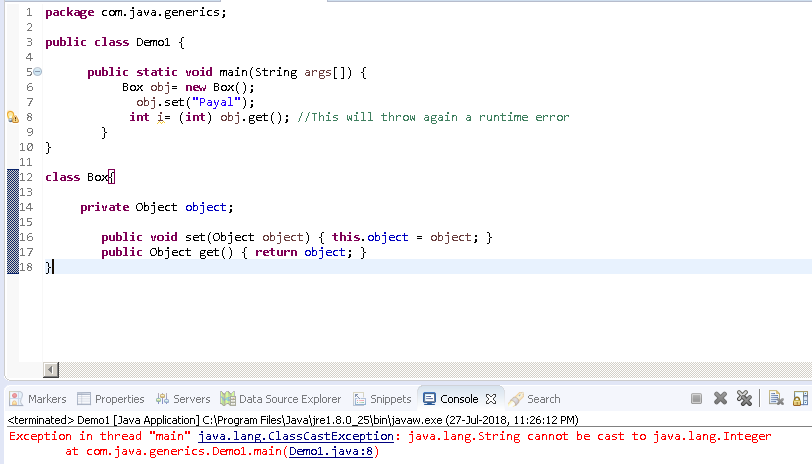
**class** Box{

**private** Object object;

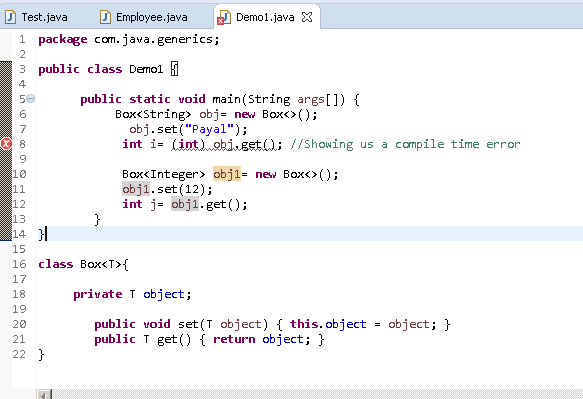
**public** **void** set(Object object) { **this**.object = object; }

**public** Object get() { **return** object; }}

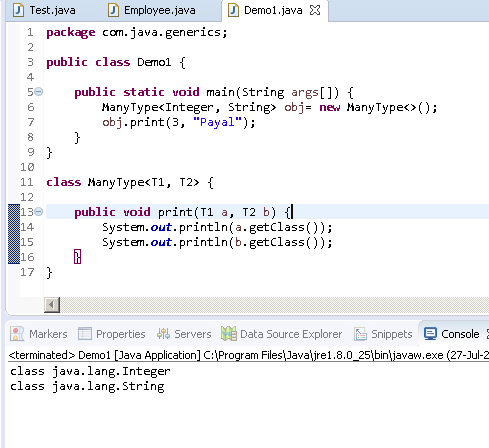
If I call a set method passing Integer value, I can do a get for a String type. So again chances of runtime errors.



By making Box class generic, we can avoid such runtime errors:



We can pass as many type parameters as we want



**Can use any name for the type:**

The 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

Though can use any variable name for type,by convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable [naming](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html#naming) conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name

## 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<>();

# 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<>();

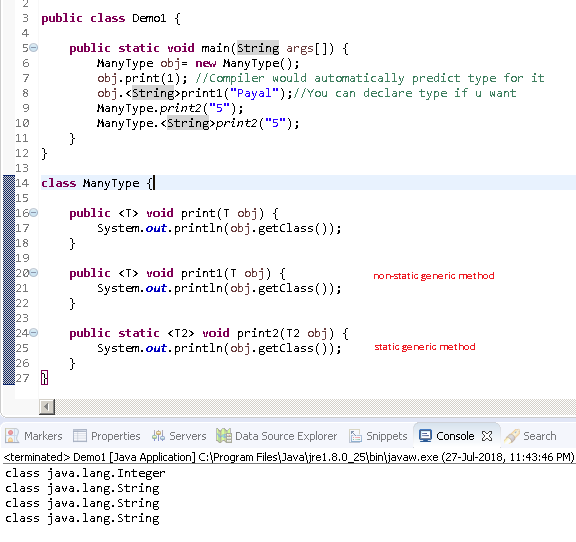
If the actual type argument is omitted, you create a raw type of Box<T>:

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.**

# 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.



Q) Convert this class to be using generics such that I can pass any kind of List, but it should be defined while creating the object.

public class Demo1 {

public static void main(String args[]) {

List<Integer> list= new ArrayList<>();

list.add(4);

ManyType1 obj= new ManyType1(list);//If list passed is of type integer, should prohibit passing string as a parameter to remove method of ManyType1 class

obj.add(4);

obj.remove("Payal"); //should not allow this. Will give runtime error

}

}

class ManyType1{

List list;

public ManyType1(List list){//List should accept all types of list: arrayList/ LinkedList/ Vector

this.list= list;

}

//To accept only List types

public void printLength() {

System.*out*.println(list.size());

}

public void add(Object obj) {

list.add(obj);

}

public void remove(Object obj) {

list.remove(obj);

}

public List getList() {

return list;

}

}

**Solution:**

class ManyType<E>{

List<E> list;

public ManyType(List<E> list){

this.list= list;

}

//To accept only List types

public void printLength() {

System.out.println(list.size());

}

public void add(E obj) {

list.add(obj);

}

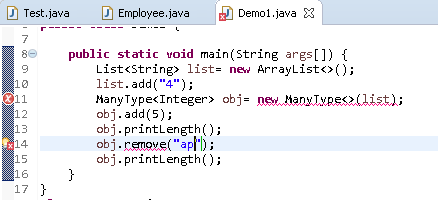
public void remove(E obj) {

list.remove(obj);

}

}

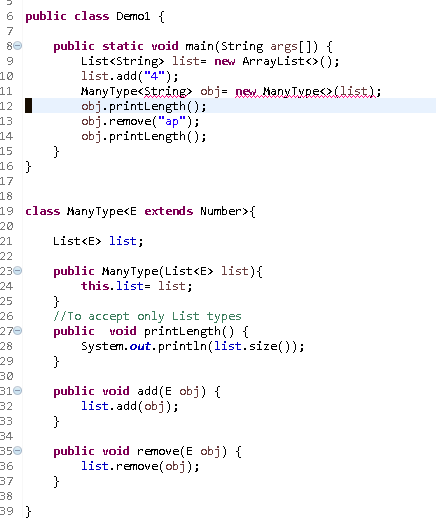
Will give u compile time errors if doing something wrong:



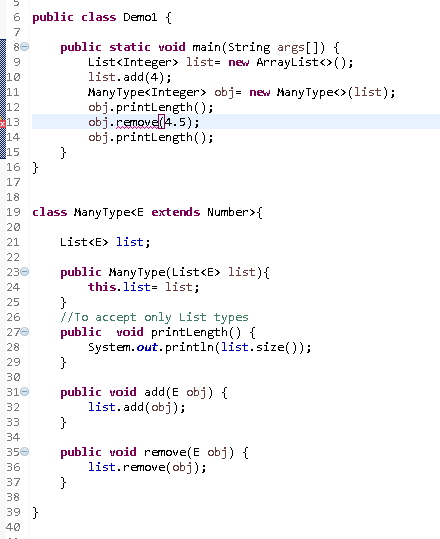
**Bounded Type Parameters:**

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

If in the above code, we want it to accept only list of Numbers, be it List<Double> or List<Integer> but not List<String>, we can restrict it using bounded type



Now it will accept only Number type List and since we are using generics for class level, we cannot differ the param type for diff. methods bcoz we are defining the type when we are creating the object.



In this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

## 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 { /\* ... \*/ }

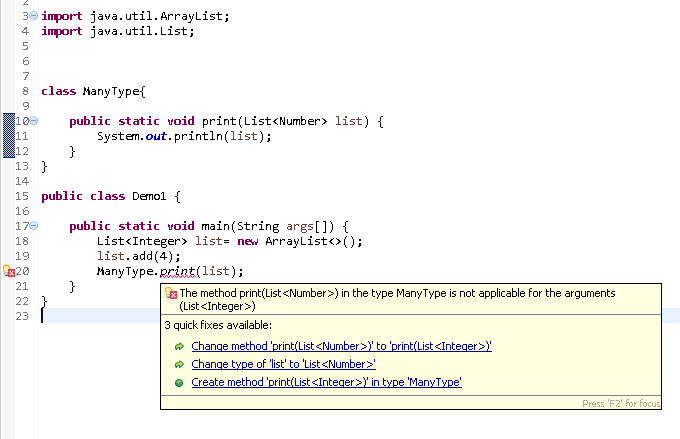
interface C { /\* ... \*/ }

class D <T extends A & B & C> { /\* ... \*/ }

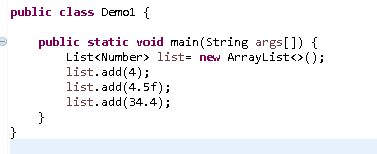
If bound A is not specified first, you get a compile-time error:

class D <T extends B & A & C> { /\* ... \*/ } // compile-time error

**You cannot use parent-child relationship for generics**

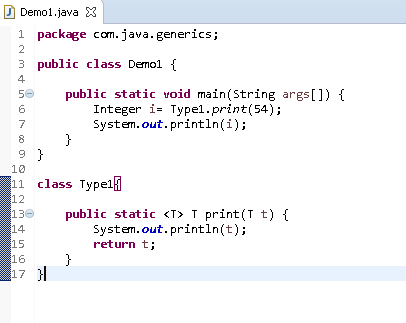


**But to a list which accepts Number, u can add integer or double values to it**

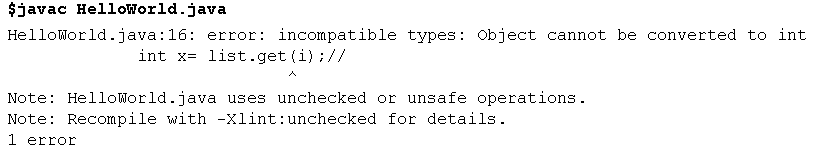


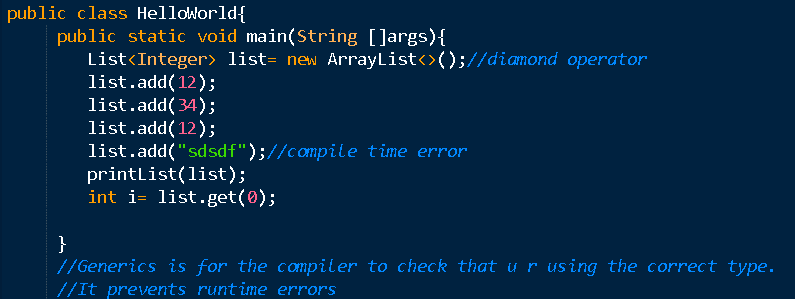
# Type Inference

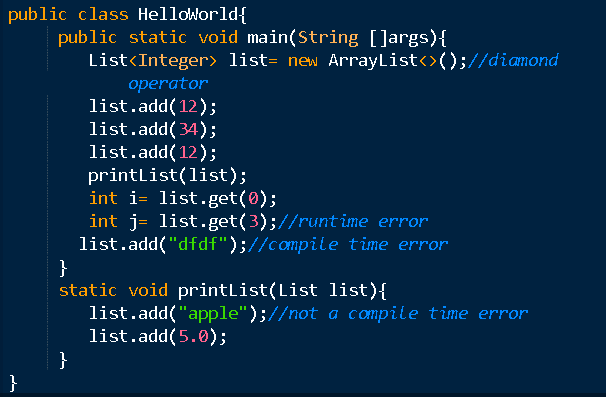
Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the most specific type that works with all of the arguments













import java.util.\*;

import java.io.\*;

public class HelloWorld{

public static void main(String []args){

Math1 math1= new Math1();

math1.add(10, 20);

Math1 math2= new Math1();

math2.add(10.0, 20.0);

Math2<Integer> math3= new Math2();

math3.add(10, 20);

Math2<Double> math4= new Math2();

math4.add(10.0, 20.0);

}

}

class Math1{

public void add(Number a, Number b){

System.out.println(a+""+ b);

}

}

//Comparable

class Math2<T extends Number & Comparable & Serializable>{//number & subclass of number and

//those which implement Comparable

public void add(T a, T b){

System.out.println(a+""+ b);

}

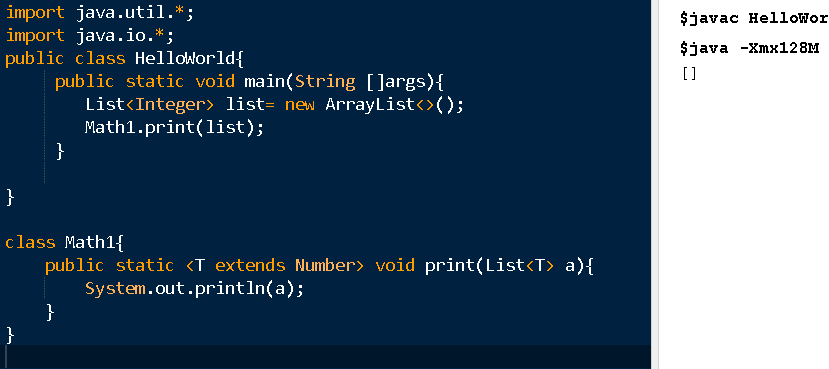
}

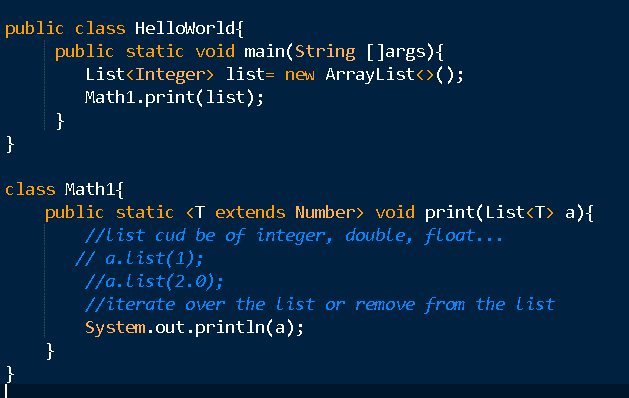
/\*class Math2{

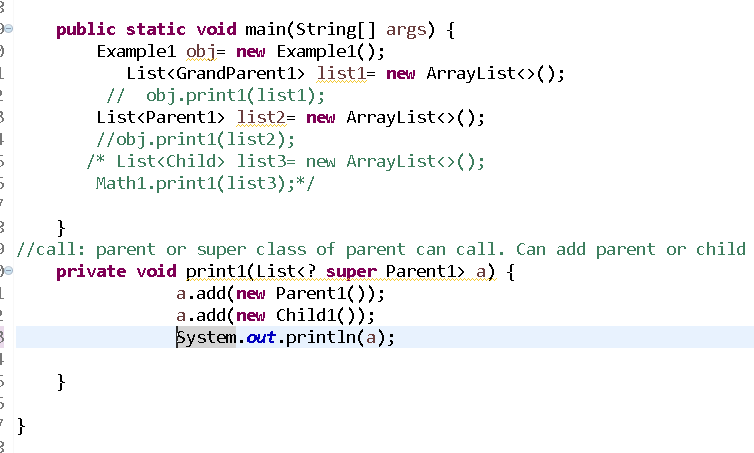
public void add(Number a, Number b){

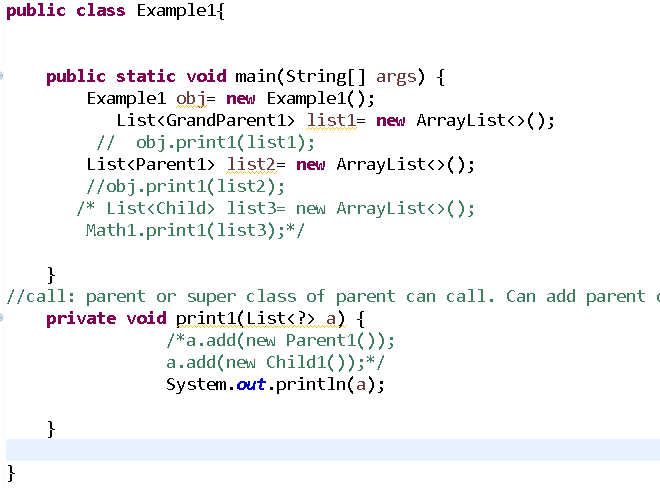
System.out.println(a+""+ b);

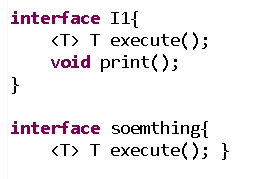
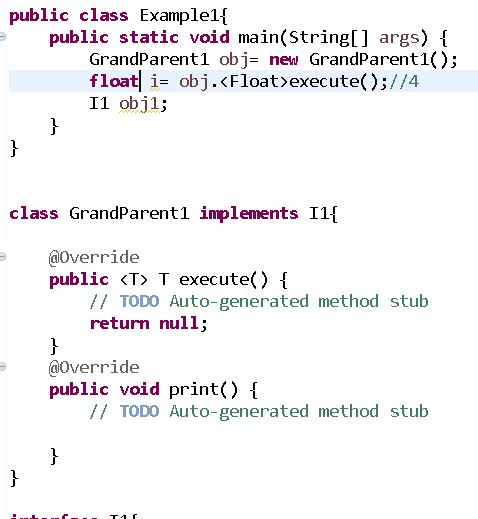
}











**You can’t have static field of type**

You can not define a static generic parameterized member in your class. Any attempt to do so will generate compile time error: Cannot make a static reference to the non-static type T.

|  |
| --- |
| public class GenericsExample<T>  {     private static T member; //This is not allowed  } |

**b) You can not create an instance of T**

Any attempt to create an instance of T will fail with error: Cannot instantiate the type T.

|  |
| --- |
| public class GenericsExample<T>  {     public GenericsExample(){        new T();     }  } |

**c) Generics are not compatible with primitives in declarations**

Yes, it’s true. You can’t declare generic expression like List or Map<String, double>. Definitely you can use the wrapper classes in place of primitives and then use primitives when passing the actual values. These value primitives are accepted by using auto-boxing to convert primitives to respective wrapper classes.

|  |
| --- |
| final List<int> ids = new ArrayList<>();    //Not allowed    final List<Integer> ids = new ArrayList<>(); //Allowed |

**d) You can’t create Generic exception class**

Sometimes, programmer might be in need of passing an instance of generic type along with exception being thrown. This is not possible to do in Java.

|  |
| --- |
| // causes compiler error  public class GenericException<T> extends Exception {} |

When you try to create such an exception, you will end up with message like this: The generic class GenericException may not subclass java.lang.Throwable.

* Generics doesn’t support sub-typing, so List<Number> numbers = new ArrayList<Integer>();will not compile, learn [why generics doesn’t support sub-typing](https://www.journaldev.com/1330/java-collections-interview-questions-and-answers#generics-sub-typing).
* We can’t create generic array, so List<Integer>[] array = new ArrayList<Integer>[10] will not compile, read [why we can’t create generic array?](https://www.journaldev.com/1330/java-collections-interview-questions-and-answers#generics-array).

<https://nofluffjuststuff.com/magazine/2016/09/time_to_really_learn_generics_a_java_8_perspective>

One of the challenges of working on a language like Java is that it needs to support years of backward compatibility. When generics were added to the language, the decision was made to remove them during the compilation process. That way no new classes are created for parameterized types, so there is no runtime penalty to using them.

Since all of that is done under the hood, all you really need to know is that at compile time:

* Bounded type parameters are replaced with their bounds
* Unbounded type parameters are replaced with Object
* Type casts are inserted where needed
* Bridge methods are generated to preserve polymorphism