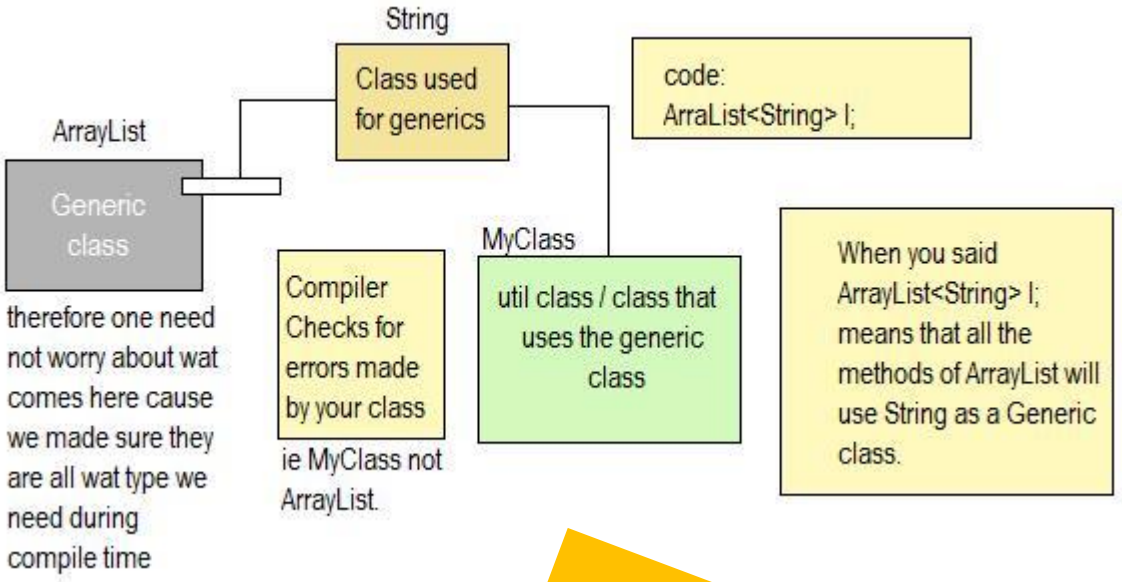
Generics



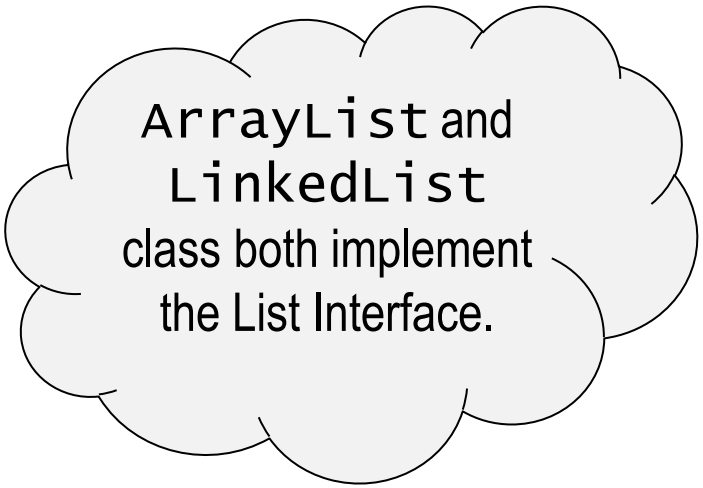
Generic Methods

Generic Classes

Generic Interfaces

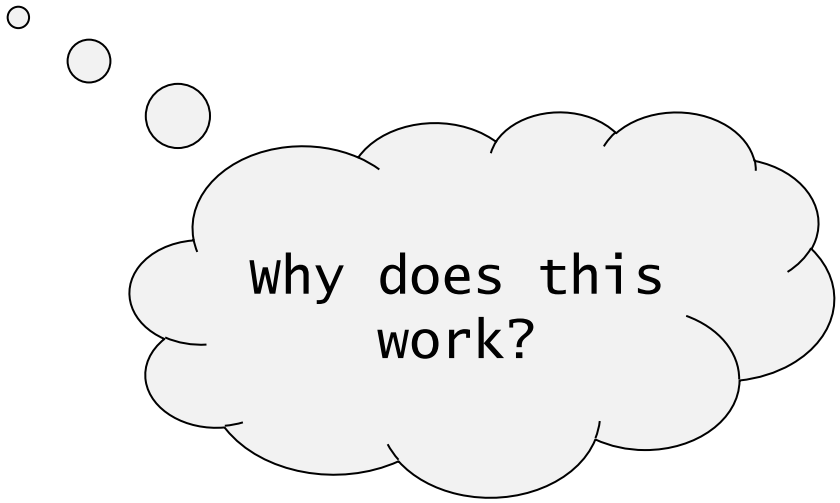
Consider this…

```
{

    List list1 = new ArrayList();
    List list2 = new LinkedList();

}
```



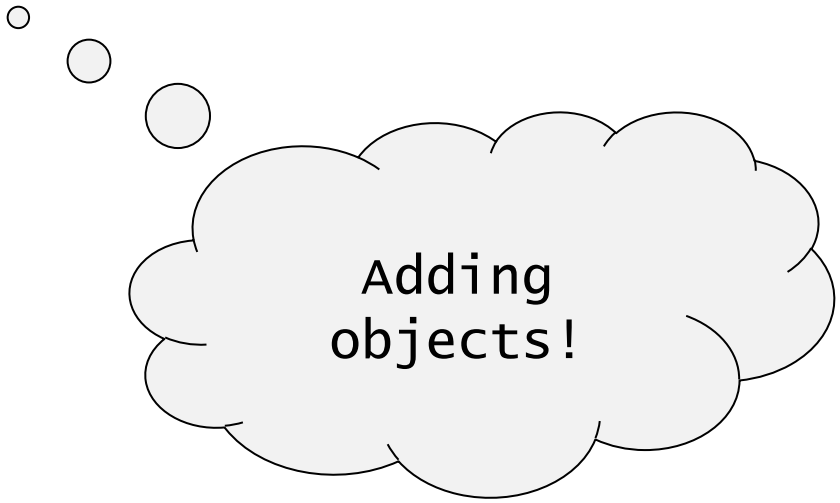
Consider this…

```
{

    List list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );


}
```

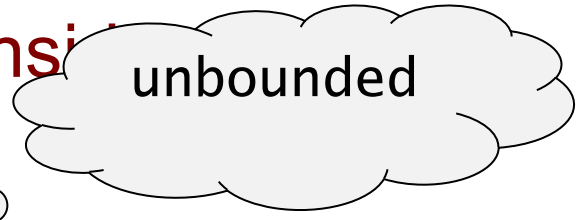


Consider this…

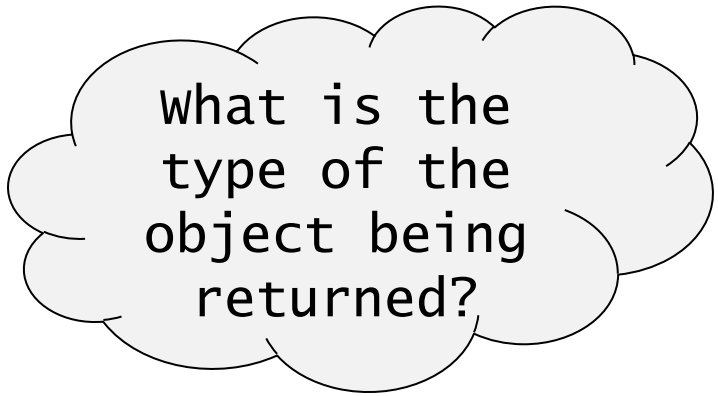
```
{

    List list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );


}
```



Consi




```
{
    List list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    ? item = list1.get(..);
}
```



Consider this…

```
{

    List<Object> list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    Object item = list1.get(..);
}
```

Consider this…

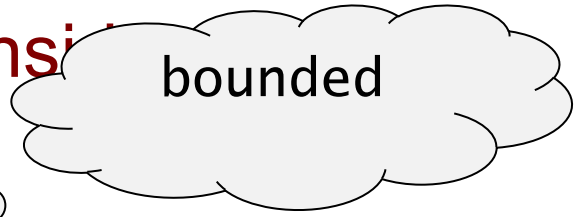
```
{

    List<Object> list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    Student item = list1.get(..);
}
```

Consider this…

```
{

    List<Object> list1 = new ArrayList();
    List list2 = new LinkedList();

    list1.add( new Integer(3) );
    list1.add( new Student() );
    list1.add( new String("some string") );

    Student item = (Student) list1.get(..);
}
```

explicit cast

Consi




```
{
    List<Student> list1 = new ArrayList();
    List list2 = new LinkedList();

    // list1.add( new Integer(3) );
    list1.add( new Student() );
    // list1.add( new String("some string") );

    Student item = list1.get(..);
}
```

Mechanism of Abstraction

- *Abstraction by Parameterization*
- Abstraction by Specification

Abstraction by Parameterization

*Abstraction by Parameterization seeks generality by allowing the same **function** to be adapted to many different contexts by providing it with the varying information on that context in a form of parameters.*

We do not write code that works on specific values, we write functions. Functions describe a computation that works on all acceptable values of the appropriate types. We specify those values in the form of parameters. Thus, the detail of what specific values are to be used is removed.

Parameterized types (e.g. Generics) are another example of abstraction by parameterization, although there the parameters are types rather than values.

Abstraction by Parameterization

Abstraction by Parameterization seeks generality by allowing the same function to be adapted to many different contexts by providing it with the varying information on that context in a form of parameters.

We do not write code that works on specific values, we write functions. Functions describe a computation that works on all acceptable values of the appropriate types. We specify those values in the form of parameters. Thus, the detail of what specific values are to be used is removed.

Parameterized types (e.g. ***Generics****) are another example of abstraction by parameterization, although there the parameters are* ***types*** *rather than* ***values****.*

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

```
 /* precondition: s must contain a character array,
  *    delimited by the null character
  * postcondition: returns the length of s as an
  *    integer;
  */

int strlen( String[] s ) {

   // Implementation is irrelevant
   return(length);
}
```

Abstraction by Specification

The specification is a contract between the client and the class.

It tells the client what can be relied upon when calling the function or method. The client should not assume anything about the behavior or implementation of the method. The specification also dictates to the developer of the method what behavior must be provided and the developer must meet the specification.

The Public Interface of the class

Principle of Abstraction

- *Abstraction by Parameterization*
- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural*
  - *data*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - ***procedural*** *// methods, parameters and returns*
  - *data*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural* *// methods, parameters and returns*
  - ***data*** *// classes and objects*
  - *iteration*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural* *// methods, parameters and returns*
  - *data* *// objects*
  - ***Iteration*** *// collections*

Principle of Abstraction

- *Abstraction by Parameterization*

- *Abstraction by Specification*
  - *Modifiability*
  - *Locality*

- *Abstraction by parameterization and abstraction by specification are powerful methods for program construction. The enable us to define three different kinds of abstraction:*
  - *procedural* *// methods, parameters and returns*
  - *data* *// objects*
  - *Iteration* *// collections*
  - ***type*** *// generics*

Generics

- Generics is the capability to parameterize datatypes.
- Allows us to define a *method* (or a *class* or an *interface*) with a *generic type* that the compiler will substitute with a concrete type.

Type Parameter Naming Conventions

Generic parameter names typically are single, uppercase letters. The most commonly used type parameter names are:

- **E - Element**
- K - Key
- N - Number
- **T - Type**
- V - Value

Generics

- Generics is the capability to parameterize datatypes.
- Allows us to define a *method* (or a *class* or an *interface*) with a *generic type* that the com… concrete type.

This makes sense as you don't want to confuse a generic type with a variable name!

Type Parameter Nam…

Generic parameter names t… uppercase letters. The most commonly used type parameter names are:

- **E - Element**
- K - Key
- N - Number
- **T - Type**
- V - Value

Generic Methods

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };


        print( strings );

    }

    public static void print( String[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };


        print( strings );

    }

    public static void print( String[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers ); // No, print is expecting a String
    }

    public static void print( String[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

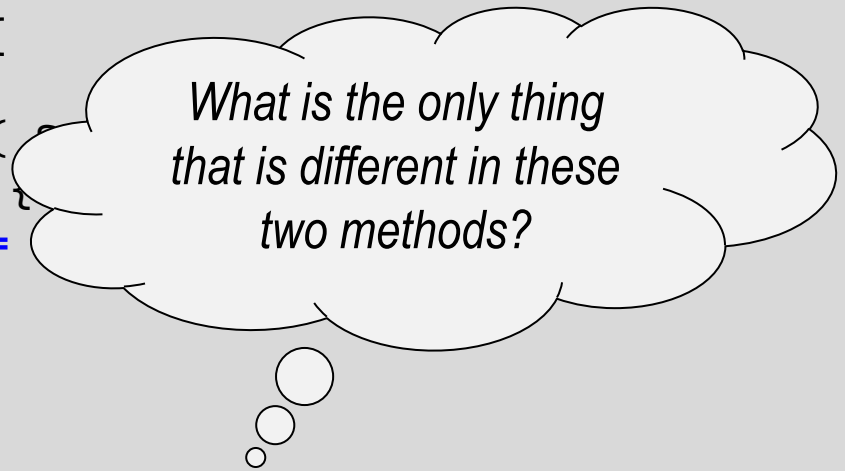
overload the method?

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers );
    }

    public static void print( String[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }

    public static void print( Integer[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

overload the method?

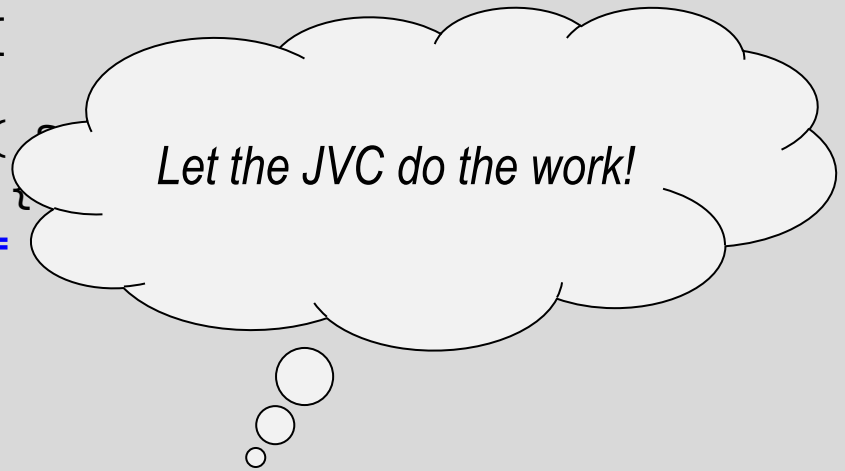
```
public class GenericMethod {

    public static void main( S
        String [] strings = {
        Integer[] integers =

        print( strings );
        print( integers );
    }

    public static void print( String[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }

    public static void print( Integer[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```



Generic Methods:

overload the method?

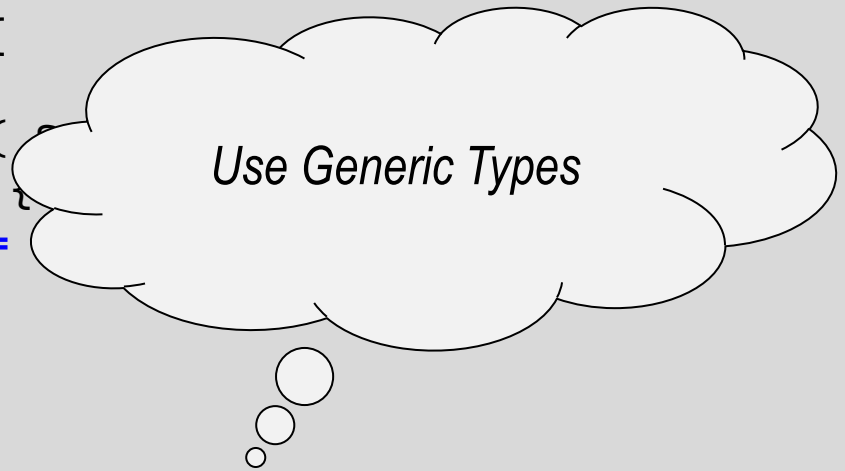
```
public class GenericMethod {

    public static void main( 
        String [] strings = {
        Integer[] integers =

        print( strings );
        print( integers );
    }

    public static void print( String[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }

    public static void print( Integer[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```



Generic Methods:

define a generic method

```
public class GenericMethod {

    public static void main( S
        String [] strings = {
        Integer[] integers =

        print( strings );
        print( integers );
    }

    public static <T> void print( T[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```



Generic Methods:

define a generic method

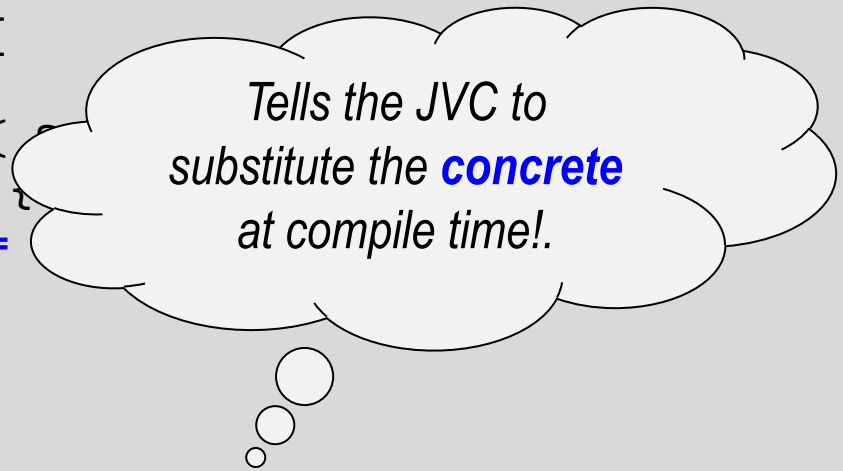
```
public class GenericMethod {

    public static void main( S
        String [] strings = {
        Integer[] integers =

        print( strings );
        print( integers );
    }

    public static <T> void print( T[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Indicates that this is a generic method.

Generic Methods:

define a generic method

```
public class GenericMethod {

    public static void main( 
        String [] strings = {
        Integer[] integers =

        print( strings );
        print( integers );
    }

    public static <T> void print( T[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```



Generic Methods:

define a generic method

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers );
    }

    public static <T> void print( T[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

define a generic method

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers );
    }

    public static <T> void print( String [] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

define a generic method

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers );
    }

    public static <T> void print( T[] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

define a generic method

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers );
    }

    public static <T> void print( Integer [] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

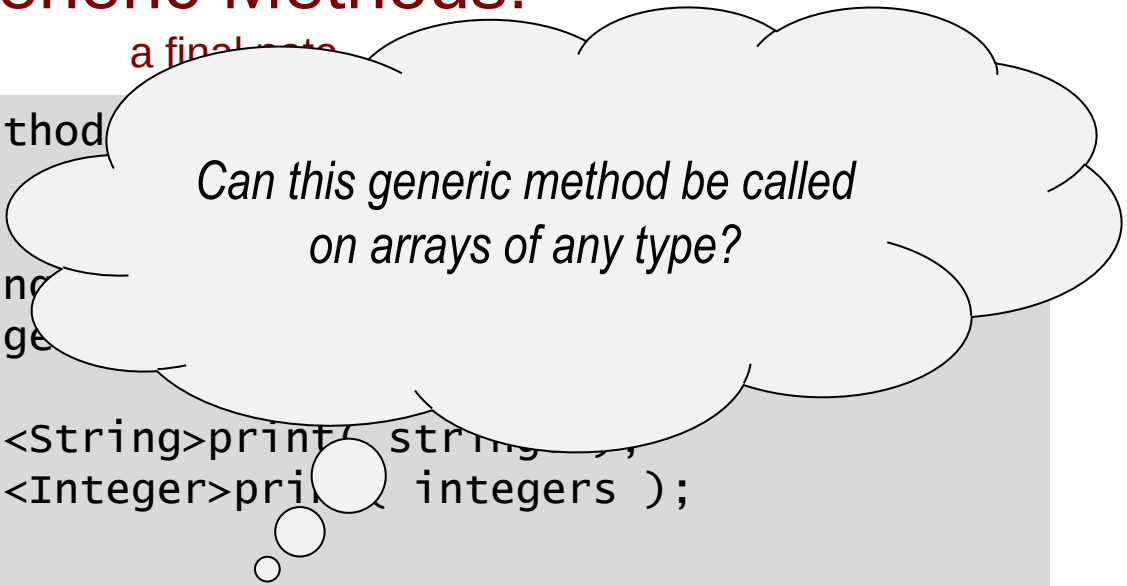
a final note

```
public class GenericMethod {

    public static void main( String [] argv ) {
        String [] strings = { "Boston", "Chicago", "NYC" };
        Integer[] integers = {1, 2, 3, 4, 5};

        print( strings );
        print( integers );
    }

    public static <T> void print( Integer [] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```

Generic Methods:

a final note

```
public class GenericMethod

    public static void
        String [] string
        Integer[] intege

        GenericMethod.<String>print( string );
        GenericMethod.<Integer>pri( integers );
    }

    public static <T> void print( T [] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i] + " " );
    }
}
```



Generic Methods:

a fir

```
public class GenericMethod {

    public static void main
        String [] strings
        Integer[] integers =

        GenericMethod.<String>print( strings );
        GenericMethod.<Integer>print( integers );
    }

    public static <T> void print( T [] list ) {
        for ( int i = 0; i < list.length; i++ )
                System.out.println( list[i].toString() + " " );
    }
}
```

*On any array of **objects.** All objects inherit the `toString()` method from Java's `Object` class. And even if not overridden, a method call can be made!*

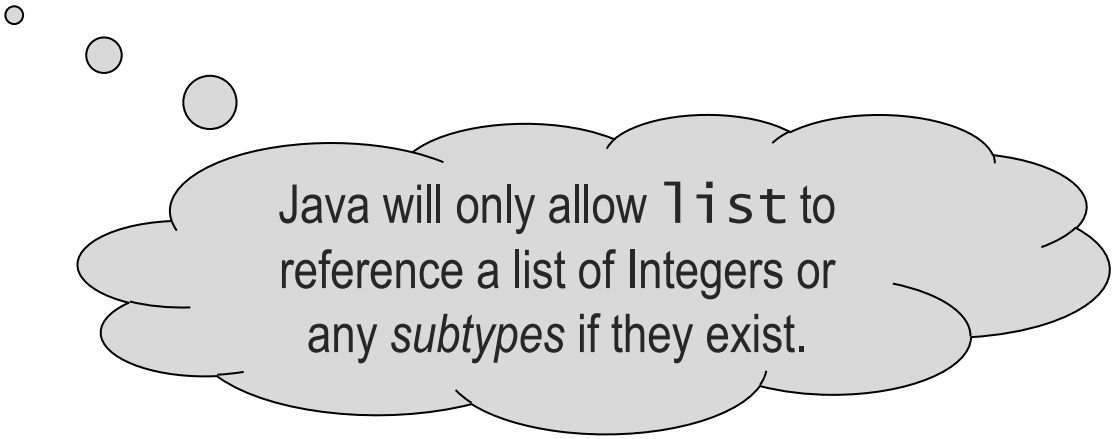
Generic Types in Java

At the core of generics is the concept of “type safety”.

A guarantee by the compiler that if correct Types are used in correct places then there should **not be** any `ClassCastException` in runtime.

Example:

List<Integer> **list;**



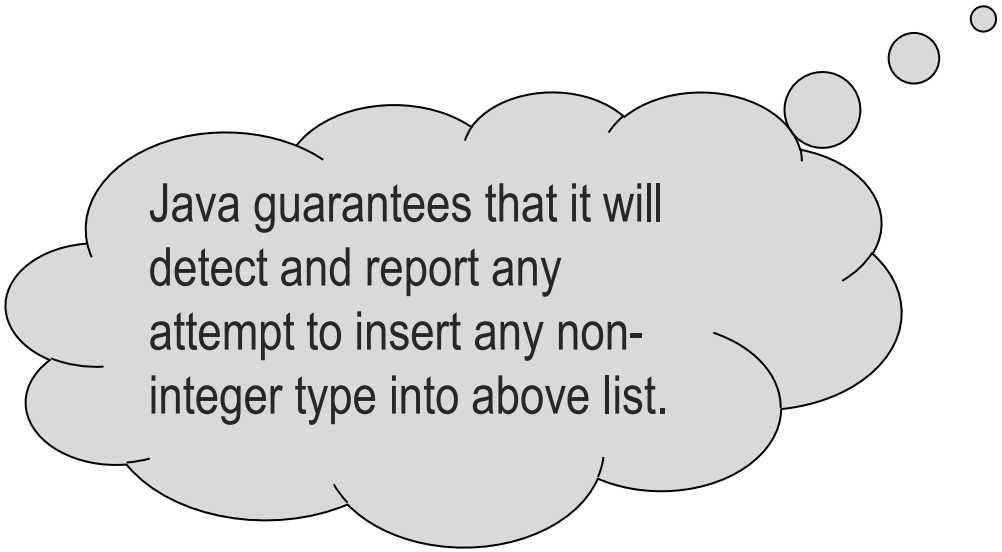
Generic Types in Java

At the core of generics is the concept of “type safety”.

A guarantee by the compiler that if correct Types are used in correct places then there should **not be** any `ClassCastException` in runtime.

Example:

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



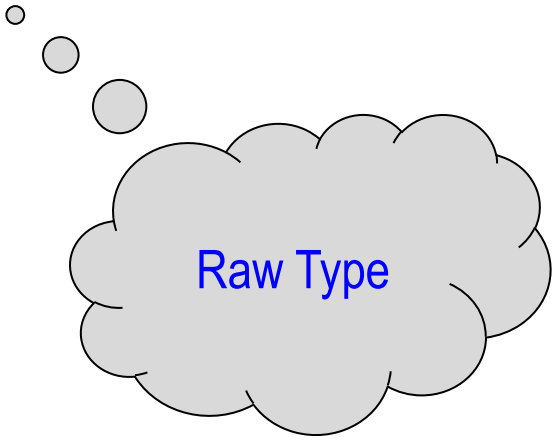
Generic Types in Java

At the core of generics is the concept of “type safety”.

A guarantee by the compiler that if correct Types are used in correct places then there should **not be** any `ClassCastException` in runtime.

Example:

List list = new List<Integer>();



Generic Types in Java

At the core of generics is the concept of "type safety".

A guarantee by the compiler that if correct Types are used in correct places then there should **not be** any `ClassCastException` in runtime.

Example:

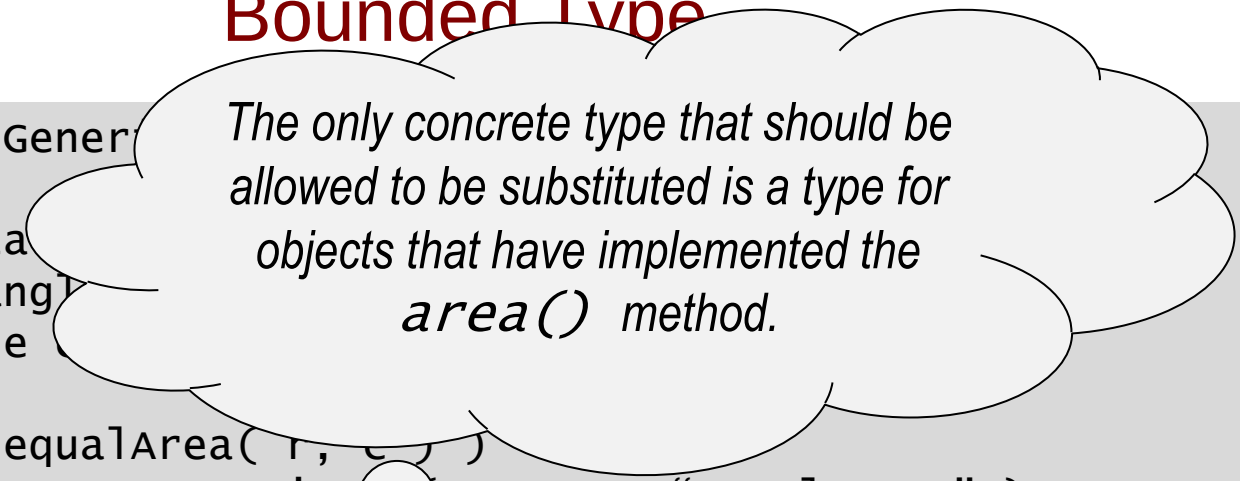
```
{
   List list = new List<Integer>();

   someMethod( list );
}
```


```
public static void someMethod( List list ) {

}
```

Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(2, 2);
        Circle c = new Circle(2);

        if ( equalArea( r, c ) )
           System.out.println( r + c + “equal area” ):
    }

    public static <T> boolean equalArea( T obj1, T obj2 ) {
        return obj1.area() == obj2.area();

    }
}
```

Bounded Type




```
public class Gener

    public sta
        Rectangl
        Circle (

        if ( equalArea( r, c ) )
           System.out.print(  r + c + "equal area" ):
    }

    public static <T> boolean equalArea( T obj1, T obj2 ) {
        return obj1.area() == obj2.area();

    }
}
```

Bounded Type

```
public class Generi

    public sta
        Rectang
        Circle

        if ( equalArea( r, c ) )
           System.out.print( r + c + "equal area" ):
    }

    public static <T extends Shape> boolean
      equalArea( T obj1, T obj2 ) {
        return obj1.area() == obj2.area();
    }
}
```

The bounded generic type specifies that T is a generic subtype of Shape and you must invoke this method with two instances of Shapes.

Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(2, 2);
        Circle c = new Circle(2);

        if ( equalArea( r, c ) )
           System.out.println( r + c + “equal area” ):
    }

    public static boolean
      equalArea( Shape obj1, Shape obj2 ) {
        return obj1.area() == obj2.area();
    }
}
```

Bounded Type

```
public class Generi

    public sta
        Rectang
        Circle

        if ( equalArea( r, c ) )
           System.out.print( r + c + “equal area” ):
    }

    public static Shape
      equalArea( Shape obj1, Shape obj2 ) {
        return obj1.area() == obj2.area();
    }
}
```

What if you wanted consistency between the type of the object passed to the method and the type of the object returned?

Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r1 = new Rectangle(2, 2);
        Rectangle r2 = new Rectangle(5, 8);

        Rectangle r3 = someMethod( r1, r2 );

    }

    public static Shape
      someMethod( Shape obj1, Shape obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```

Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r1 = new Rectangle(2, 2);
        Rectangle r2 = new Rectangle(5, 8);

        Rectangle r3 = someMethod( r1, r2 );

    }

    public static Shape
      someMethod( Shape obj1, Shape obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```

Bounded Type

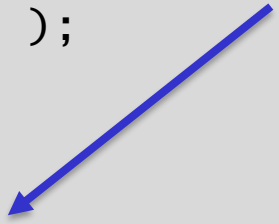
```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r1 = new Rectangle(2, 2);
        Rectangle r2 = new Rectangle(5, 8);

        Rectangle r3 = (Rectangle) someMethod( r1, r2 );

    }

    public static Shape
      someMethod( Shape obj1, Shape obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```

Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Circle c1 = new Circle(2);
        Circle c2 = new Circle(4);;

        Circle c3 = (Circle) someMethod( c1, c2 );

    }

    public static Shape
      someMethod( Shape obj1, Shape obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```

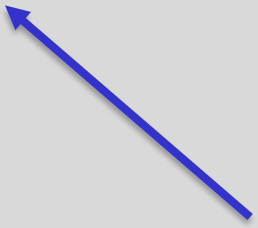
Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Circle c1 = new Circle(2);
        Circle c2 = new Circle(4);;

        Circle c3 = someMethod( c1, c2 );

    }

    public static <T extends Shape> T
      someMethod( T obj1, T obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```



Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(5,4);
        Circle c = new Circle(4);;

        ?  x = someMethod( r, c );

    }

    public static <T extends Shape> T
      someMethod( T obj1, T obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```



Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(5,4);
        Circle c = new Circle(4);;

        Circle x = someMethod( r, c );

    }

    public static <T ext
      someMethod( T obj1,
        ...
        ...

        return (obj1);
    }
}
```

Compiler error… inferred type does not conform to upper bound.

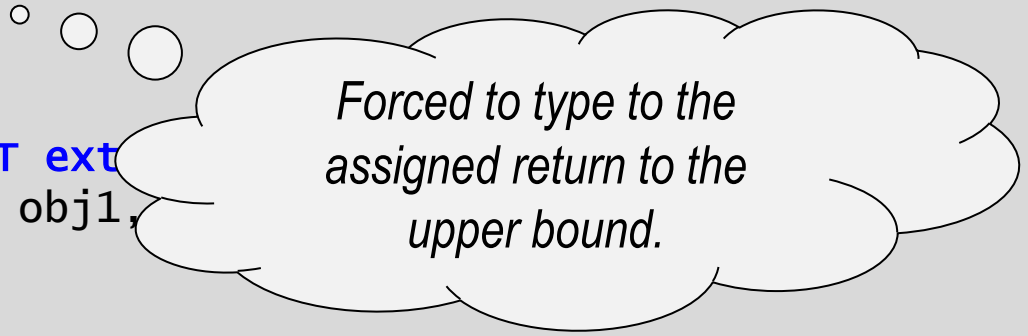
Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(5,4);
        Circle c = new Circle(4);;

        Rectangle x = someMethod( r, c );

    }

    public static <T ext
      someMethod( T obj1,
        ...
        ...

        return (obj1);
    }
}
```

Compiler error… inferred type does not conform to upper bound.

Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(5,4);
        Circle c = new Circle(4);;

        Shape x = someMethod( r, c );

    }

    public static <T ext
      someMethod( T obj1,
        ...
        ...

        return (obj1);
    }
}
```



Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Circle c1 = new Circle(5);
        Circle c2 = new Circle(4);;

        Circle x = someMethod( c1, c2 );

    }

    public static <T extends Shape> T
      someMethod( T obj1, T obj2 ) {
        ...
        ...

        return (obj1);
    }
}
```

Un-Bounded Type

```
public class GenericMethod_2 {

    public static void main( String [] argv ) {
        Rectangle r = new Rectangle(2, 2);
        Circle c = new Circle(2);

        if ( equalArea( r, c ) )
           System.out.println( r + c + "equal area" ):
    }

    public static <T> boolean
      equalArea( T obj1, T obj2 ) {
        return obj1.area() == obj2.area();
    }
}
```

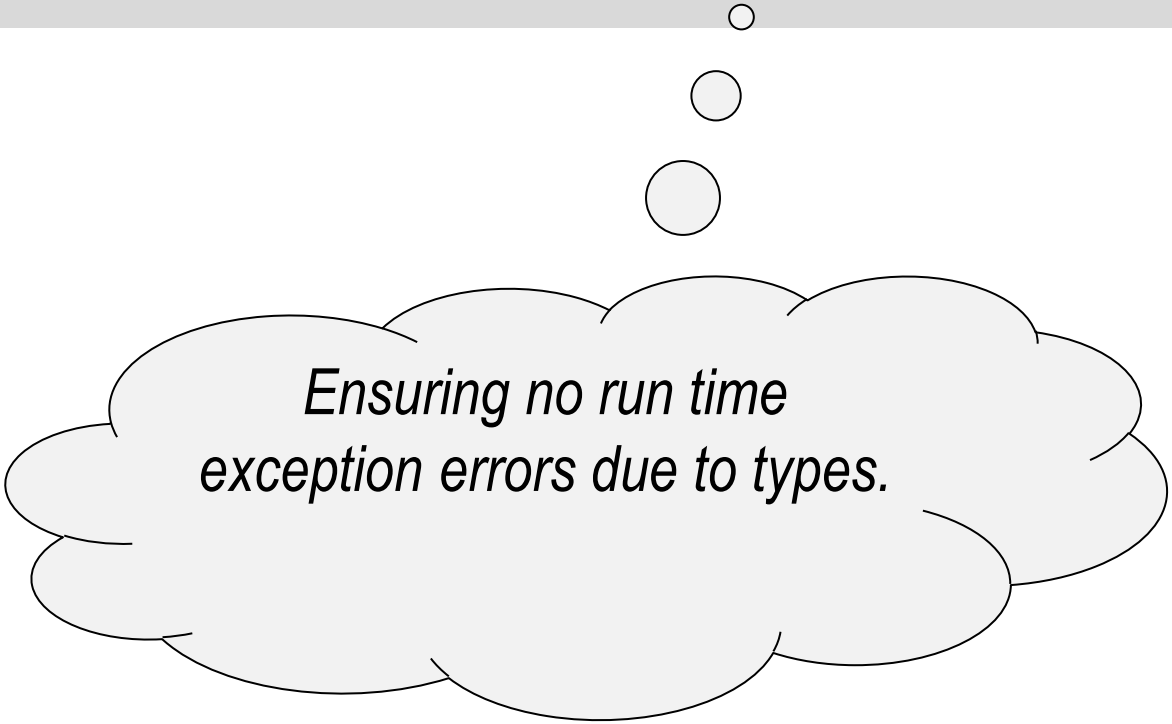
Bounded Type

```
public class GenericMethod_2 {

    public static
        Rectan
        Circle

        if ( equalArea(
           System.out.println
    }

    public static <T extends Object> boolean
      equalArea( T obj1, T obj2 ) {
        return obj1.area() == obj2.area();
    }
}
```

The unbounded generic type by default is bounded to the `Object` class!

Generics

- By using *bounded types* we can specify allowable types and objects that the class or method can work with. If you attempt to use an incompatible type, the compiler can detect it.

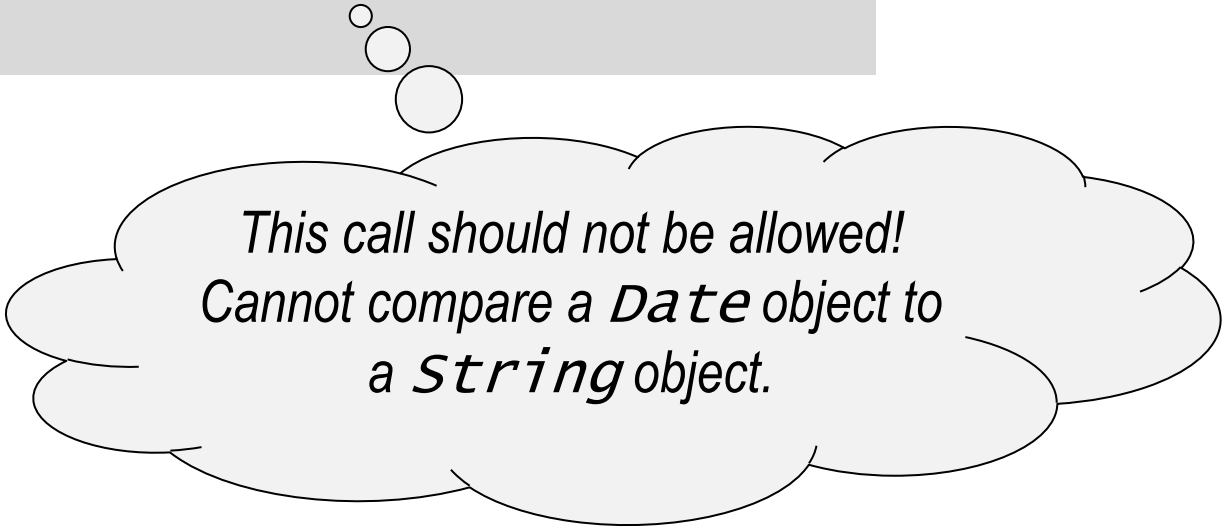


Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable {
    public int compareTo( Object o );
}
```

```
public class testClass {
    public static void main( String [] a ) {
        Date d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```

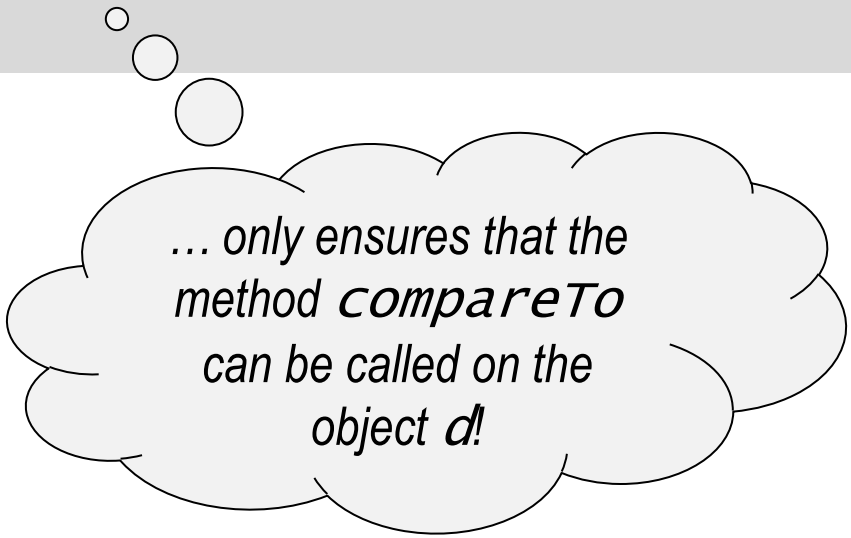


Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable {
    public int compareTo( Object o );
}
```

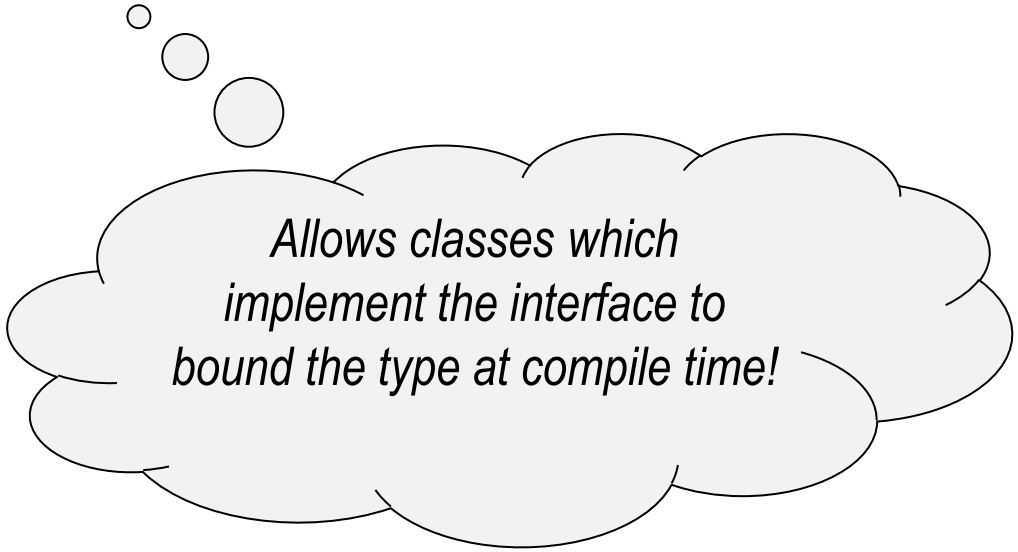
```
public class testClass {
    public static void main( String [] a ) {
        Comparable d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```



Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```



Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable {
    ...

    public int compareTo( Object other ) {
            ...
    }

}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<object>
{
    ...

    public int compareTo( Object other ) {
            ...
    }

}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<Date> {
    ...

    public int compareTo( Date other ) {
            ...
    }

}
```

```
public class testClass {
    public static void main( String [] a ) {
        Date d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<Date> {
    ...

    public int compareTo( Date other )
           ...
    }

}
```

Compiler can now detect that we are passing an incompatible type to the `compareTo` method of the `Date` class

```
public class testClass {
    public static void main( String[] a ) {
        Date d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<Date> {
    ...

    public int compareTo( Date other )
           ...
    }

}
```

Compiler can again detect that we are passing an incompatible type to the `compareTo` method of the `Date` class!

```
public class testClass {
    public static void main( String[] a ) {
        Comparable<Date> d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<Date> {
    ...

    public int compareTo( Date other )
           ...
    }

}
```

There is no compiler error here, but we would not be able to invoke any other method of the Date class on object d.

```
public class testClass {
    public static void main( String[] a ) {
        Comparable<Date> d = new Date( 7, 13, 2019 );

        if ( d.compareTo( new Date() ) )
           ...
}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<Date> {
    ...

    public int compareTo( Date other ) {
            ...
    }

}
```

```
public class testClass {
    public static void main( String [] a ) {
        Comparable d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```

Generic Interfaces

revisiting the Comparable interface

```
public interface Comparable<T> {
    public int compareTo( T o );
}
```

```
public class Date implements Comparable<Date> {
    ...

    public int compareTo( Date other )
           ...
    }

}
```

Potential run-time error. There is a compiler warning, but the Java code would compile (string is an Object).

```
public class testClass {
    public static void main( String[] a ) {
        Comparable<Object> d = new Date( 7, 13, 2019 );

        if ( d.compareTo( "red" ) )
           ...
}
```

Generic Classes

the ArrayList class

- The Java ArrayList class allows you to create *array* lists of objects that are not restricted to the limitations of primitive (fixed) Java arrays.
- ArrayList is a Java Class.

```
+ArrayList()
+add(o: Object)
+clear(): void
+contains(o: Object): boolean
+get(index: int): Object
+indexOf(o Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+remove(index: int): boolean
+size(): int
+set(index: int, o: Object): Object
```

Generic Classes

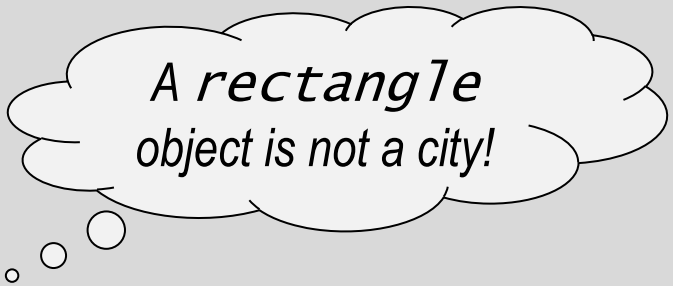
the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList cityList = new ArrayList();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston" );
        cityList.add("NYC");
        cityList.add("Athens");
        cityList.add("Beijing");
        cityList.add("Seoul");

        cityList.remove( "NYC" );




    }

}
```

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList cityList = new ArrayList();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston" );
        cityList.add("NYC");
        cityList.add("Athens");
        cityList.add("Beijing");
        cityList.add("Seoul");

        cityList.remove("NYC");

        cityList.add( new Rectangle() );



    }

}
```



Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList cityList = new ArrayList();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston");
        cityList.add("NYC");
        cityList.add("Athens");
        cityList.add("Beijing");
        cityList.add("Seoul");

        cityList.remove("NYC");

        cityList.add( new Rectangle() );



    }

}
```

But my list allows it because it is a list of objects!

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<Object> cityList = new ArrayList<Object>();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston");
        cityList.add("NYC");
        cityList.add("Athens");
        cityList.add("Beijing");
        cityList.add("Seoul");

        cityList.remove("NYC");

        cityList.add( new Rectangle() );



    }

}
```

This is the default type for ArrayLists!

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<String> cityList = new ArrayList<String>();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston" );
        cityList.add("NYC");
        cityList.add("Athens");
        cityList.add("Beijing");
        cityList.add("Seoul");

        cityList.remove("NYC");

        cityList.add( new Rectangle() );



    }

}
```

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<String> cityList = new ArrayList<String>();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston" );
        cityList.add("NYC");
        cityList.add("Athens");
        cityList.add("Beijing");
        cityList.add("Seoul");

        cityList.remove("NYC");

        cityList.add( new Rectangle() ); // compiler error!



    }

}
```

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<City> cityList = new ArrayList<City>();

        // Create a list of cities
        cityList.add("London");
        cityList.add("Boston");
        cityList.add("NYC");
        cityList.add("Ath
        cityList.add("Bei
        cityList.add("Seoul");

        cityList.remove("NYC");









    }

}
```

Assume a class `City`, create an array list of `City` objects!

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<City> cityList = new ArrayList<City>();

        // Create a list of cities
        cityList.add( new City("London") );
        cityList.add( new City("Boston") );
        cityList.add( new City("NYC") );
        cityList.add( new City("Athens") );
        cityList.add( new City("Beijing") );
        cityList.add( new City("Seoul") );

        cityList.remove("NYC"); // would this work?




    }

}
```

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<City> cityList = new ArrayList<City>();

        // Create a list of cities
        cityList.add( new City("London") );
        cityList.add( new City("Boston") );
        cityList.add( new City("NYC") );
        cityList.add( new City("Athens") );
        cityList.add( new City("Beijing") );
        cityList.add( new City("Seoul") );

        cityList.remove( new City("NYC") );




    }

}
```

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<City> cityList = new ArrayList<City>();

        // Create a list of citi
        cityList.add( new City
        cityList.add( new Ci
        cityList.add( new Cit
        cityList.add( new City(
        cityList.add( new City("Beijing") );
        cityList.add( new City("Seoul") );

        cityList.remove( new City("NYC") );




    }

}
```

Assuming that the class `City` *has overridden the* `equals` *method!*

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<City> cityList = new ArrayList<City>();

        // Create a list of cities
        cityList.add( new City("London") );
        cityList.add( new City("Boston") );
        cityList.add( new City("NYC") );
        cityList.add( new City("Athens") );
        cityList.add( new City("Beijing") );
        cityList.add( new City("Seoul") );

        cityList.remove( new City("NYC") );

        cityList.add("NYC"); // compile time error


    }

}
```

Generic Classes

the ArrayList class

```
public class TestArrayList {

    public static void main( String [] argv ) {

        ArrayList<City> cityList = new ArrayList<City>();

        // Create a list of cities
        cityList.add( new City("London") );
        cityList.add( new City("Boston") );
        cityList.add( new City("NYC") );
        cityList.add( new City("Athens") );
        cityList.add( new City("Beijing") );
        cityList.add( new City("Seoul") );

        cityList.remove( new City("NYC") );

        cityList.add(new Rectangle()); // compile time error


    }

}
```

Generic Classes

the ArrayList class

- The Java ArrayList is a Generic class in Java (beginning with version 1.5 of Java).

```
+ArrayList()
+add(o: E): void
+clear(): void
+contains(o: E): boolean
+get(index: int): Object
+indexOf(o E): int
+isEmpty(): boolean
+lastIndexOf(o: E): int
+remove(o: E): boolean
+remove(index: int): boolean
+size(): int
+set(index: int, o: E): E
```

A Generic Stack Class

Limiting a Stack to Objects of a Given Type

- An interface for a Stack class.

```
public interface Stack {
    boolean push(Object item);
    Object pop();
    Object peek();
    boolean isEmpty();
    boolean isFull();
}
```

- Allows me to implement a Stack of any type of Object!

Limiting a Stack to Objects of a Given Type

- A *generic* interface and class.
- Here's a generic version of our `Stack` interface:

```
public interface Stack<T> {
    boolean push(T item);
    T pop();
    T peek();
    boolean isEmpty();
    boolean isFull();
}
```

- It includes a *type variable* `T` in its header and body.
  - used as a placeholder for the actual type of the items

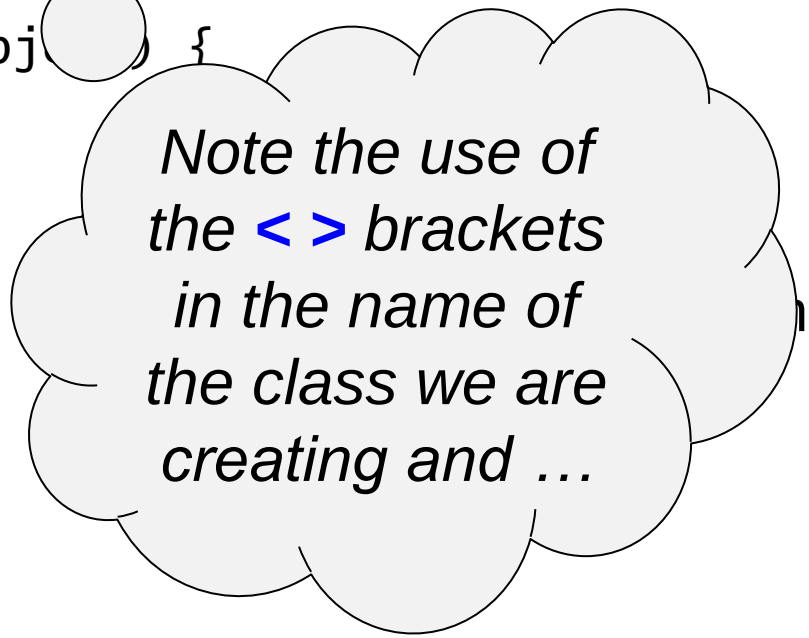
A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as a placeholder for the actual type of the items.

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // dex of the top item
    …
    public boolean push(T obj ) {
        …
    }
    …
}
```

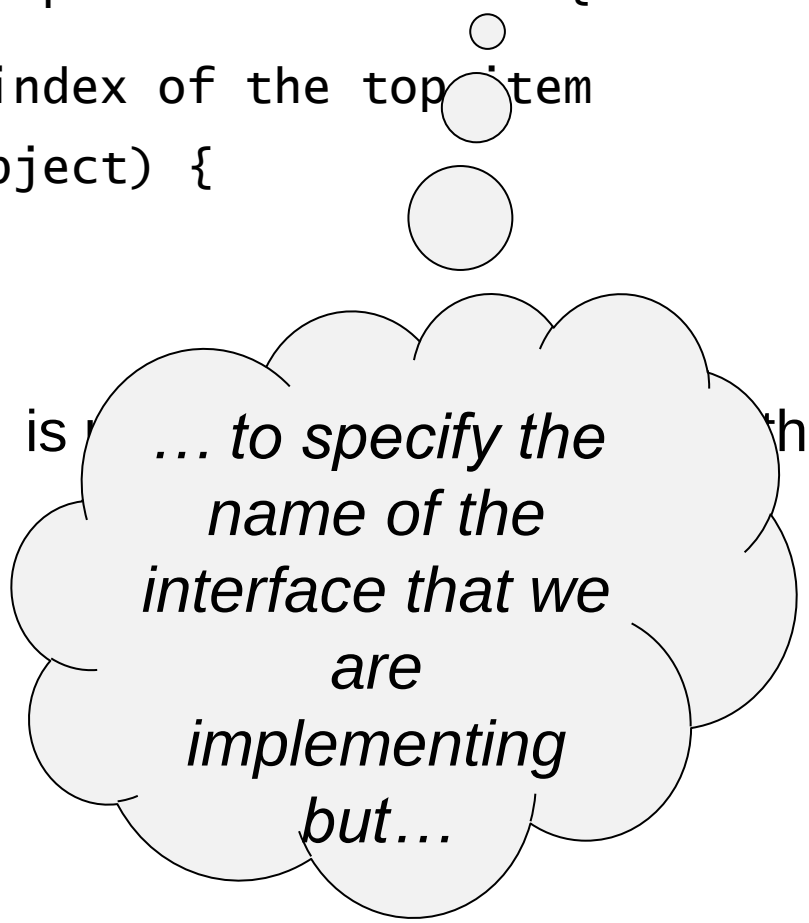


- Once again, a type variable T ne actual type of the items.

A Generic `ArrayStack` Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

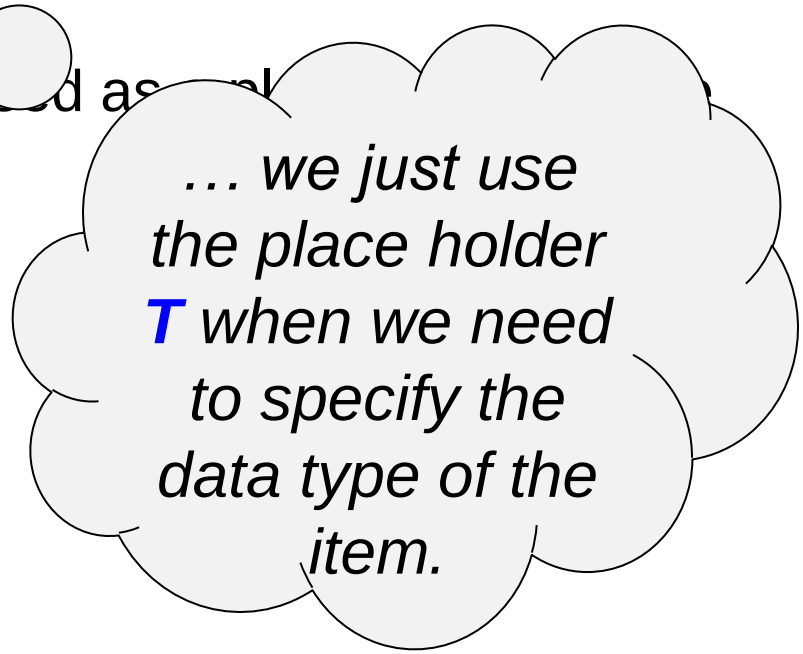
- Once again, a type variable `T` is … the actual type of the items.



A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …
    public boolean push(T object) {
        …
    }
    …
}
```

- Once again, a type variable `T` is used as [illegible] actual type of the items.

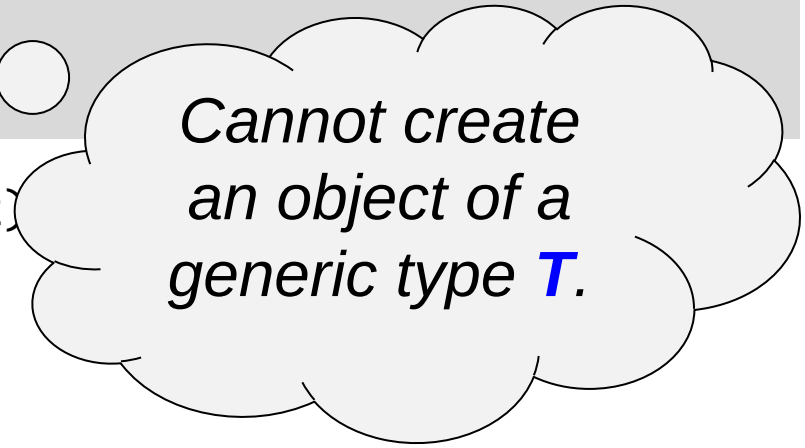


A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …

    public ArrayStack(int maxSize) {
        items = new T[maxSize];  // Java does not all this
        top = -1;
    }

    public boolean push(T object) {
        …
    }
    …
}
```

A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;     // index of the top item
    …

    public ArrayStack(int maxSize) {
        items = new T[maxSize];  // Java does not all this
        top = -1;
    }

    public boolean push(T object)
        …
    }
    …
}
```



Generic Types in Java

An array is a collection of similar type of elements.

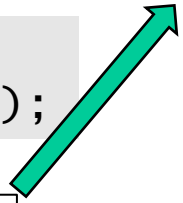
Arrays preserve their type information in runtime, but generics use ***type erasure*** and remove any type information in runtime.

As such, instantiating a generic array in java is not permitted.

If Java Generics worked like C++

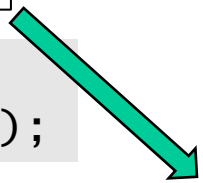
```
public class ArrayStack<String> {
    private String[] items;
    private int top;
    ...
    public boolean push(String item) {
        ...
```

```
ArrayStack<String> s1 =
  new ArrayStack<String>(10);
```



```
public class ArrayStack<T> ... {
    private T[] items;
    private int top;
    ...
    public boolean push(T item) {
        ...
```

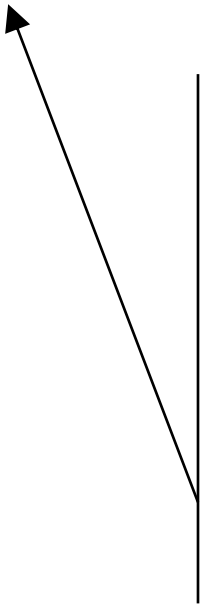
```
ArrayStack<Integer> s1 =
  new ArrayStack<Integer>(25);
```



```
public class ArrayStack<Integer> {
    private Integer[] items;
    private int top;
    ...
    public boolean push(Integer item) {
        ...
```

Generic Types in Java

- Java implements generics using "*type erasure*".
- Essentially all the extra information added using generics into source code is removed from the bytecode generated from it.



The Java compiler uses the generic type information to compile the code but then erases it afterward. As a result, the information is not available at run-time.

Essentially the compiler uses the type information to confirm that a generic type is used safely, but then converts to the Raw type.

Generic Types in Java

- Java implements generics using "*type erasure*".
- Essentially all the extra information added using generics into source code is **removed** from the bytecode generated from it.
- After translation by type erasure, all information regarding type parameters and type arguments has disappeared.
- All instantiations of the same generic type share the same runtime type, namely the **raw type**.

Generic Types in Java

- Java implements generics using "*type erasure*".
- Essentially all the extra information added using generics into source code is **removed** from the bytecode generated from it.
- After translation by type erasure, all information regarding type parameters and type arguments has disappeared.
- All instantiations of the same generic type share the same runtime type, namely the **raw type**.

Essentially the compiler *hides* all information related to type parameters and type arguments. Example:

`List<String>`, `List<Long>` are translated as type **`List`** in the bytecode.

Generic

- Java imp
- Esse ics
int ted
fro
- Afte egarding
type p peared.
- All instantiations of the s c type share the same runtime type, nam the **raw type**.

When compiling a class or interface that extends or implements a parameterized interface, the compiler may need to do specific casting or create a synthetic method, called a *bridge method*, as part of the type erasure process.

Essentially the com ler *hides* all information related to type parameters and type arguments. Example:

`List<String>`, `List<Long>` are translated as type `List` in the bytecode.

Example

```
{
   ArrayList<String> list = new ArrayList<String>();

    list.add("someString");

    String s = list.get(0);
}
```

post-compilation…

```
{
   ArrayList list = new ArrayList();

    list.add("someString");

    String s = (String) (list.get(0));
}
```

Example:

unbounded generic type

```
public static <T> void print( T[] arr ) {

   for ( int i = 0; i<arr.length; i++ )
      System.out.println( arr[i] );

}
```

post-compilation…

```
public static Object void print( Object[] arr ) {

   for ( int i = 0; i<arr.length; i++ )
      System.out.println( arr[i].toString() );

}
```

Example:

bounded generic type

```
public static <T extends Shape> boolean
   equalArea( T o1, T o2 ) {



}
```

post-compilation…

```
public static Shape boolean
   equalArea( Shape o1, Shape o2 ) {

   // cast as needed during compile time

}
```

Generic Types in Java

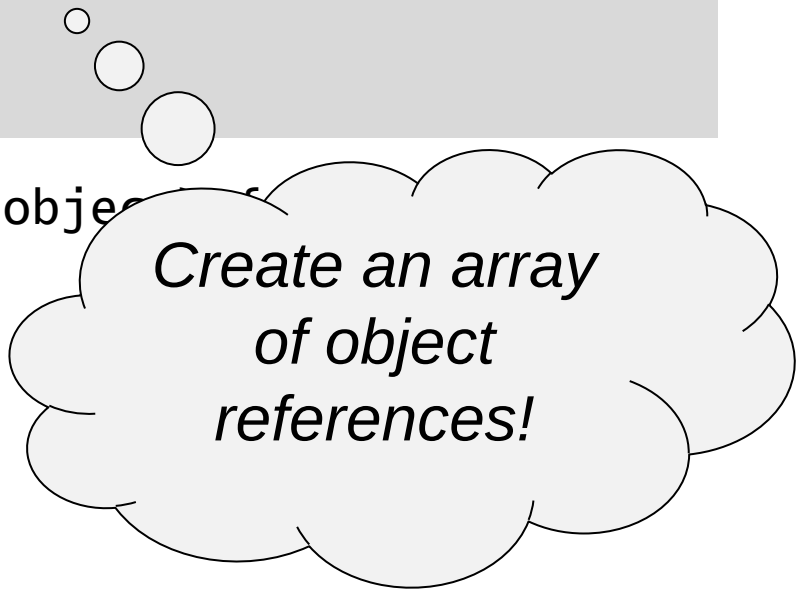
An array is a collection of similar type of elements.

Arrays preserve their type information in runtime, but generics use type erasure and remove any type information in runtime.

Because of type erasure, instantiating a generic array in Java is not permitted.

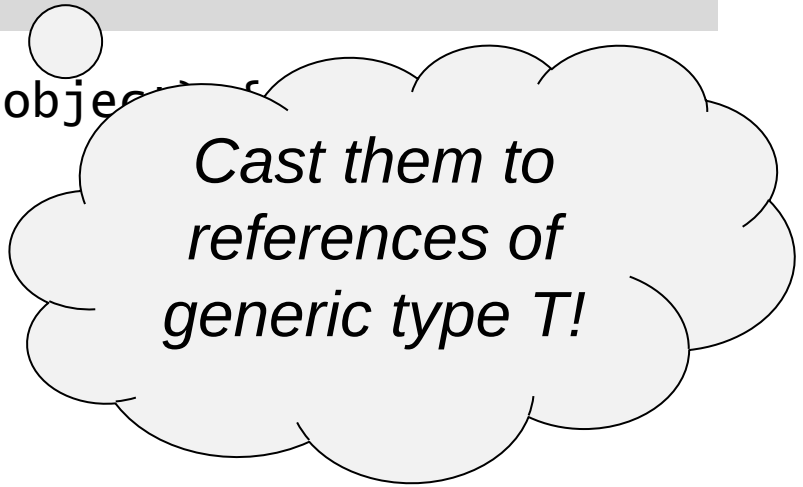
A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;     // index of the top item
    …

    public ArrayStack(int maxSize) {
        items =         new Object[maxSize];
        top = -1;
    }

    public boolean push(T obje
        …
    }
    …
}
```



A Generic ArrayStack Class

```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …

    public ArrayStack(int maxSize) {
        items = (T[]) new Object[maxSize];
        top = -1;
    }

    public boolean push(T obje
        …
    }
    …
}
```



A Generic ArrayStack Class

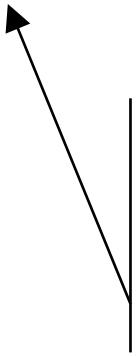
```
public class ArrayStack<T> implements Stack<T> {
    private T[] items;
    private int top;    // index of the top item
    …

    public ArrayStack(int maxSize) {
        items = (T[]) new Object[maxSize];
        top = -1;
     }

    public boolean push(T object) {
        …
    }
    …
}
```

```
ArrayStack<String> s1 = new ArrayStack<String>(10);
ArratStack<Integers> s2 = new ArrayStack<Integers>(50);

ArrayStack<Objects> s3 = new ArrayStack<Objects>(12);
```

Restrictions with Generic Types

1. new E()

Cannot create an *instance* of a Generic Type.



This statement is executed at run-time, but because of type erasure the generic type is not available at run-time.

Restrictions with Generic Types

1. new E()

   Cannot create an *instance* of a Generic Type.

2. new E[]

   Cannot create an array using a Generic Type.

```
E[] elements = (E[]) new Object[maxNum];
```

Restrictions with Generic Types

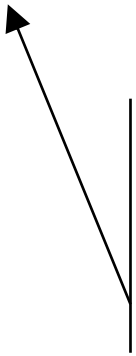
1. new E()

Cannot create an *instance* of a Generic Type.

2. new E[]

Cannot create an array using a Generic Type.

```
E[] elements = (E[]) new Object[maxNum];
```



Note: This may cause a unchecked compiler warning because the compiler cannot be certain that the casting will succeed at run-time.

Restrictions with Generic Types

1. new E()

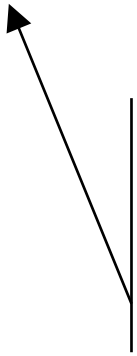
   Cannot create an *instance* of a Generic Type.

2. new E[]

   Cannot create an array using a Generic Type.

```
E[] elements = (E[]) new Object[maxNum];
```

3. A generic type parameter of a class is not allowed in a static context (within the class).



All instances of a generic class have the same run-time class, therefore static variables and methods of a generic class are shared by all of its instances.

Restrictions with Generic Types

1. new E()

Cannot create an *instance* of a Generic Type.

2. new E[]

Cannot create an array using a Generic Type.

```
E[] elements = (E[]) new Object[maxNum];
```

3. A generic type parameter of a class is not allowed in a static context (within the class).

```
public class testClass<E> {
     // cannot use generic type to declare static member
       • public static E member;
      // cannot use generic type in static method
       • public static void someMethod(E param) {
             E lvar;
          }
}
```

Restrictions with Generic Types

1. new E()

Cannot create an *instance* of a Generic Type.

2. new E[]

Cannot create an array using a Generic Type.

```
E[] elements = (E[]) new Object[maxNum];
```

3. A generic type parameter of a class is not allowed in a static context (within the class).

```
public class testClass<E> {
     // cannot use generic type to declare static member
       • public static E member;
      // cannot use generic type in static method
       • public static void someMethod(E param) {
             E lvar;
          }
}
```

Restrictions with Generic Types

4. Exception classes cannot be generic!

Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<Object> list;

        list = new ArrayList<Object>();
        list = new ArrayList<String>();
        list = new ArrayList<Integer>();
   }
}
```

Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<Object> list;

        list = new ArrayList<Object>(); // valid assignment
        list = new ArrayList<String>();
        list = new ArrayList<Integer>();
   }
}
```

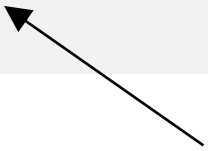
Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<object> list;

        list = new ArrayList<Object>();
        list = new ArrayList<String>();  // compiler error
        list = new ArrayList<Integer>(); // compiler error
   }
}
```

Class `String` and class `Integer` are subtypes of `Object`, but `ArrayList<String>` and `ArrayList<Integer>` are not subclasses of `ArrayList<Object>`!

Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<?> list;

        list = new ArrayList<Object>();
        list = new ArrayList<String>();
        list = new ArrayList<Integer>();
   }
}
```



Unbounded wildcard…. Which is ….

Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<? extends Object> list;

        list = new ArrayList<Object>();
        list = new ArrayList<String>();
        list = new ArrayList<Integer>();
   }
}
```

Allows all Object types and subtypes.

Bounded to the Object type, more specifically it sets the upper bound to be class `Object`.

Wildcards in Generic Types

```
public class ListTest {

    public static void main(String[] args) {
        List<? extends Number> list;

        list = new ArrayList<Integer>();
        list = new ArrayList<Float>();
        list = new ArrayList<Double>();
    }
}
```

Only allow Numeric types?

Bounded to the Number type, more specifically it sets the upper bound to be class Number.

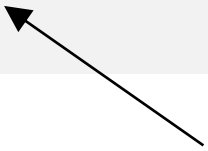
Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<? extends Number> list;

        list = new ArrayList<Integer>();
        list = new ArrayList<Float>();
        list = new ArrayList<String>(); // Compiler error
   }
}
```

Bounded to the Number type, more specifically it sets the **upper bound** to be class Number.

Wildcards in Generic Types

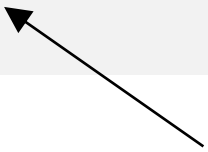
```
public class ListTest {

   public static void main(String[] args) {
        List<? super Integer> list;

        list = new ArrayList<Integer>();
        list = new ArrayList<Float>();  // compiler error
        list = new ArrayList<Double>(); // compiler error
   }
}
```



Bounded to the Integer type, more specifically sets the **lower bound** to be class Integer.

Wildcards in Generic Types

```
public class ListTest {

   public static void main(String[] args) {
        List<? super Integer> list;

        list = new ArrayList<Integer>();
        list = new ArrayList<Number>();
        list = new ArrayList<Object>();
   }
}
```



Bounded to the Integer type, more specifically sets the **lower bound** to be class Integer.

Wildcards in Generic Types

```
public class GenericsTester {

   public static <T extends Number>
                    double sum(List<T> numberlist ) {
      double sum = 0.0;
      for (Number n : numberlist)
         sum += n.doubleValue();
      return sum;
   }

   public static void main(String args[]) {
      List<Integer> integerList = Arrays.asList(1, 2, 3);
      System.out.println("sum = " + sum(integerList));

      List<Double> doubleList = Arrays.asList(1.2, 2.3, 3.5);
      System.out.println("sum = " + sum(doubleList));
   }
}
```

Need for Wildcards in Generic Types

```
public class GenericsTester {

   public static double sum(List<Number> numberlist) {
      double sum = 0.0;
      for (Number n : numberlist) sum += n.doubleValue();
      return sum;
   }

   public static void main(String args[]) {
      List<Integer> integerList = Arrays.asList(1, 2, 3);
      System.out.println("sum = " + sum(integerList));

      List<Double> doubleList = Arrays.asList(1.2, 2.3, 3.5);
      System.out.println("sum = " + sum(doubleList));
   }
}
~
```

Need for Wildcards in Generic Types

```
public class GenericsTester {

   public static double sum(List<Number> numberlist) {
      double sum = 0.0;
      for (Number n : numberlist) sum += n.doubleValue();
      return sum;
   }

   public static void main(String args[]) {
      List<Number> integerList = Arrays.asList(1, 2, 3);
      System.out.println("sum = " + sum(integerList));

      List<Number> doubleList = Arrays.asList(1.2, 2.3, 3.5);
      System.out.println("sum = " + sum(doubleList));
   }
}
~
```

Need for Wildcards in Generic Types

```
public class GenericsTester {

   public static double sum(List<? extend Number> numberlist) {
      double sum = 0.0;
      for (Number n : numberlist) sum += n.doubleValue();
      return sum;
   }

   public static void main(String args[]) {
      List<Integer> integerList = Arrays.asList(1, 2, 3);
      System.out.println("sum = " + sum(integerList));

      List<Double> doubleList = Arrays.asList(1.2, 2.3, 3.5);
      System.out.println("sum = " + sum(doubleList));
   }
}
~
```

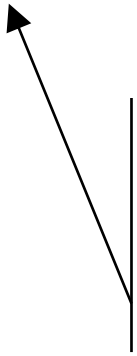
Wildcards and Generic types?

- The question mark **(?)**, called the wildcard, represents an unknown type.

- A wildcard parameterized type is an instantiation of a generic type where at least one type argument is a wildcard.

Unbounded Wildcard

- To declare an unbounded wildcard, simply use the wildcard character as <?>.
- "?" denotes any unknown type, It can represent *any* Type at in code for.



is equivalent to:

```
? extends Object
```

Unbounded Wildcard

- To declare an unbounded wildcard, simply use the wildcard character as <?>.
- "?" denotes any unknown type, It can represent *any* Type at in code for.

Example:

```
ArrayList<?> nList = new ArrayList<Number>();

nList = new ArrayList<Integer>();
nList = new ArrayList<Number>();
nList = new ArrayList<Float>();
```

Unbounded Wildcard

- To declare an unbounded wildcard, simply use the wildcard character as <?>.
- "?" denotes any unknown type, It can represent *any* Type at in code for.

Example:

```
ArrayList<?> nList = new ArrayList<Number>();

nList = new ArrayList<Integer>();
nList = new ArrayList<Number>();
nList = new ArrayList<Float>();
```

Bounded Wildcard

- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the *extends* keyword, followed by its upper bound as in **<? extends T>.**
- All Types which are either "T" or extends T means a subtype of T.

Example:

```
ArrayList<? extends Number>
    nList = new ArrayList<Number>();

nList = new ArrayList<Integer>();
nList = new ArrayList<Float>();
nList = new ArrayList<Long>();
```

Bounded Wildcard

- To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the *extends* keyword, followed by its upper bound as in **<? extends T>.**
- All Types which are either "T" or extends T means a subtype of T.

Example:

```
ArrayList<? extends Number>
    nList = new ArrayList<Number>();

nList = new ArrayList<Integer>();
nList = new ArrayList<Float>();
nList = new ArrayList<Long>();
```

Bounded Wildcard

- To declare an lower-bounded wildcard, use the wildcard character ('?'), followed by the ***super*** keyword, followed by its upper bound as in <? super T>.
- This bounds allows all types which are "T" and super classes of T.
- Example:

```
ArrayList<? super Integer>
    nList = new ArrayList<Number>();

nList = new ArrayList<Integer>();
nList = new ArrayList<Number>();
nList = new ArrayList<Float>();  // Error
```

Wildcard bound vs. Generic Type bound

Wildcard

```
? extends SuperType
? Super SubType
```

1. A wildcard can have a lower or an upper bound.

2. A wildcard can have only one bound, while a type parameter can have several bounds.

Generic Type Bound

```
T extends Class
```

Wildcard bound vs. Generic Type bound

Wildcard

```
? extends SuperType
? Super SubType
```

```
// upper bound
// lower bound
```

1. **A wildcard can have a lower or an upper bound**

2. A wildcard can have only one bound, while a type parameter can have several bounds.

Generic Type Bound

```
T extends Class
```

Wildcard bound vs. Generic Type bound

Wildcard

```
? extends SuperType
? Super SubType
```

```
// upper bound
// lower bound
```

1. A wildcard can have a lower or an upper bound.

2. A wildcard can have only one bound, while a type parameter can have several bounds.

Generic Type Bound

```
T extends Class & Interface1 & … & InterfaceN
```