WIA 1002 DATA STRUCTURE

SEM 2, SESSION 2024/205

NURUL JAPAR
nuruljapar@um.edu.my

HOO WAI LAM
wlhoo@um.edu.my

Generic

- Definition
- ArrayList
- Classes & Interfaces
- Methods
- Bounded Generic
- Raw Type
- Erasure
- Restriction on Generic

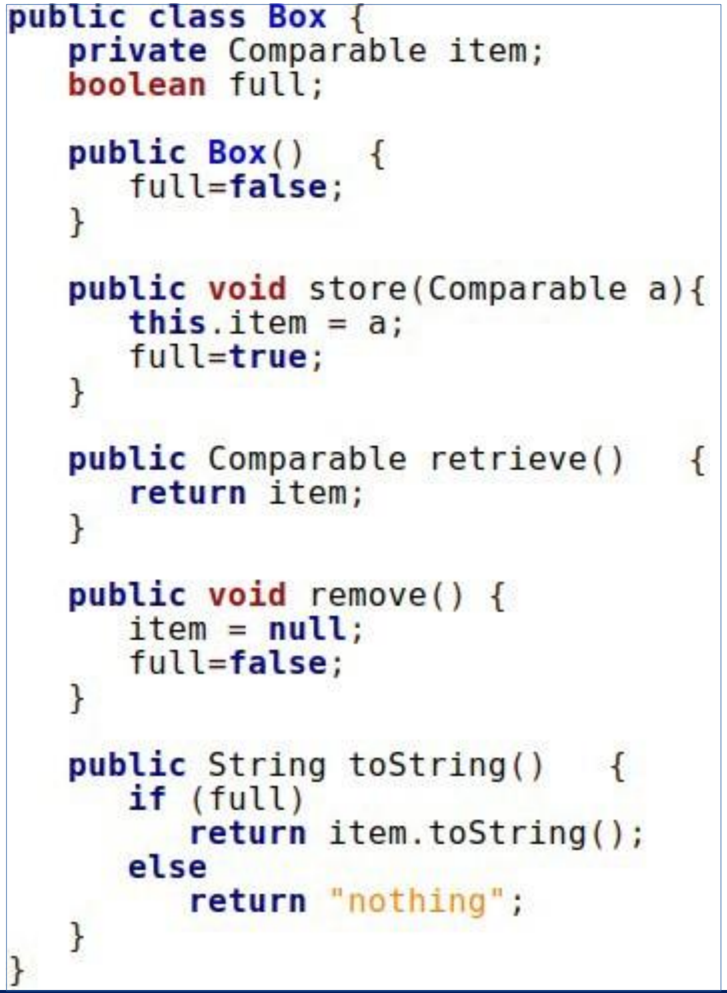
What is Generic?

- ***Generic*** is the ***capability to parameterize types***. With this capability, you can define a class or a method with generic types that can be substituted using **concrete types** by the compiler.
- For example, you may define a **generic class (e.g. "array")** that **stores the elements of a generic type**. From this generic class, you may create an "array" for **holding strings** and an "array" for **holding numbers**. Here, strings and numbers are ***concrete*** types that **replace** the ***generic*** type.
- The difference is that the inputs to formal parameters are **values**, while the inputs to type parameters are **types**.

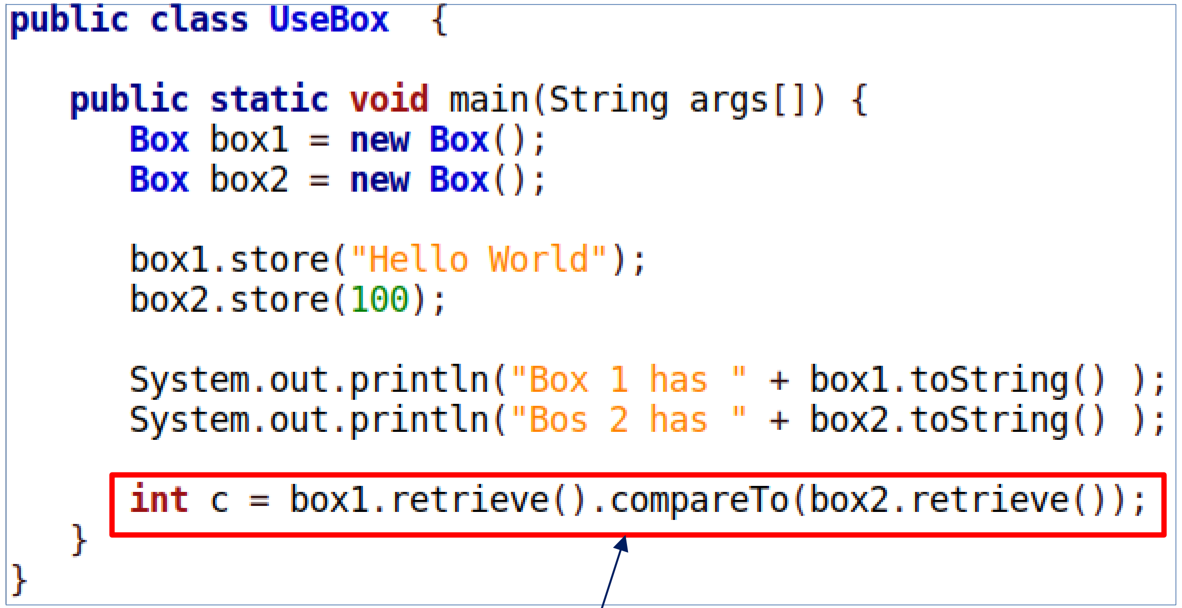
Why Generics?

- Generics in programming provide **stronger type checks at compile time** by allowing you to specify the types of objects a class or method can work with.
- This ensures that type mismatches are caught at **compile time** before the program runs, reducing runtime errors and improving code reliability.
- If you attempt to use the class or method with an **incompatible object**, a **compile error occurs**.

Why Generics?



```java
public class Box {
   private Comparable item;
   boolean full;

   public Box()   {
      full=false;
   }

   public void store(Comparable a){
      this.item = a;
      full=true;
   }

   public Comparable retrieve()   {
      return item;
   }

   public void remove() {
      item = null;
      full=false;
   }

   public String toString()   {
      if (full)
         return item.toString();
      else
         return "nothing";
   }
}
```

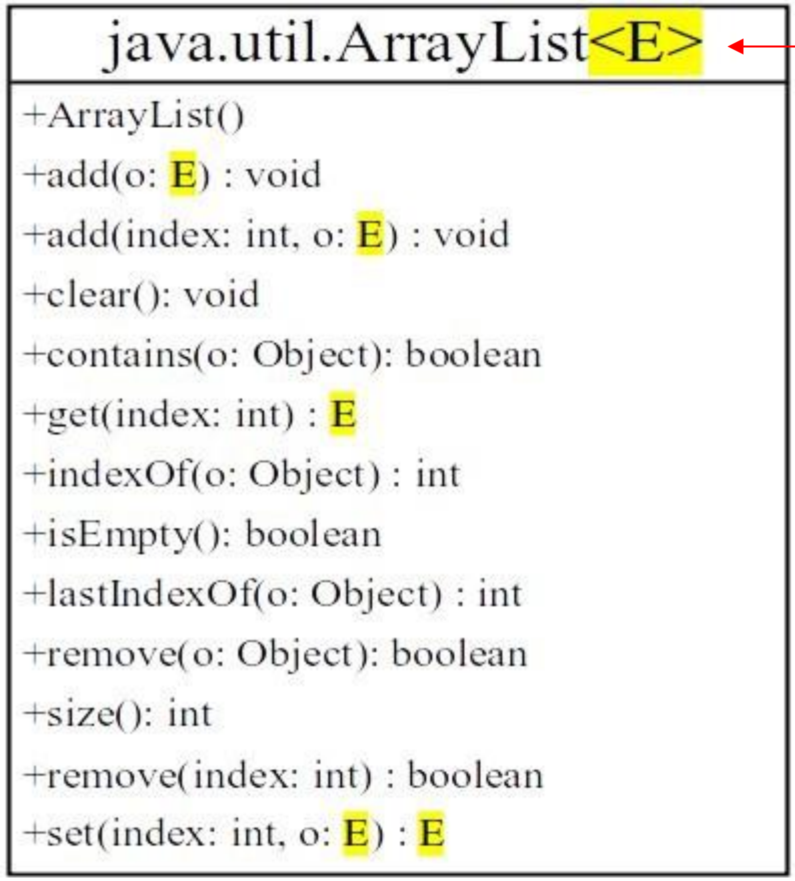


```java
public class UseBox  {

   public static void main(String args[]) {
      Box box1 = new Box();
      Box box2 = new Box();

      box1.store("Hello World");
      box2.store(100);

      System.out.println("Box 1 has " + box1.toString() );
      System.out.println("Bos 2 has " + box2.toString() );

      int c = box1.retrieve().compareTo(box2.retrieve());
   }
}
```

Compile ok, but Runtime error

The ArrayList Class

- You can create an array to store objects. But the array’s size is fixed once the array is created. Java provides the **ArrayList** class that can be used to store an unlimited number of objects.



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

Formal generic type

<E> is meant to be an Element

The ArrayList Class

java.util.ArrayList<E>

`+ArrayList()`	Creates an empty list.
`+add(o: E) : void`	Appends a new element `o` at the end of this list.
`+add(index: int, o: E) : void`	Adds a new element `o` at the specified index in this list.
`+clear(): void`	Removes all the elements from this list.
`+contains(o: Object): boolean`	Returns true if this list contains the element `o`.
`+get(index: int) : E`	Returns the element from this list at the specified index.
`+indexOf(o: Object) : int`	Returns the index of the first matching element in this list.
`+isEmpty(): boolean`	Returns true if this list contains no elements.
`+lastIndexOf(o: Object) : int`	Returns the index of the last matching element in this list.
`+remove(o: Object): boolean`	Removes the element `o` from this list.
`+size(): int`	Returns the number of elements in this list.
`+remove(index: int) : boolean`	Removes the element at the specified index.
`+set(index: int, o: E) : E`	Sets the element at the specified index.

The ArrayList Class

- **ArrayList** is known as a *generic class* with a *generic type* E. You can specify a concrete type to replace E. For example, the following syntax creates an **ArrayList** and assigns its reference to **variable** *cities*. As such, it can be used to store **strings**.
- ArrayList<String> cities = **new** ArrayList<String>();

Or, can be written as:

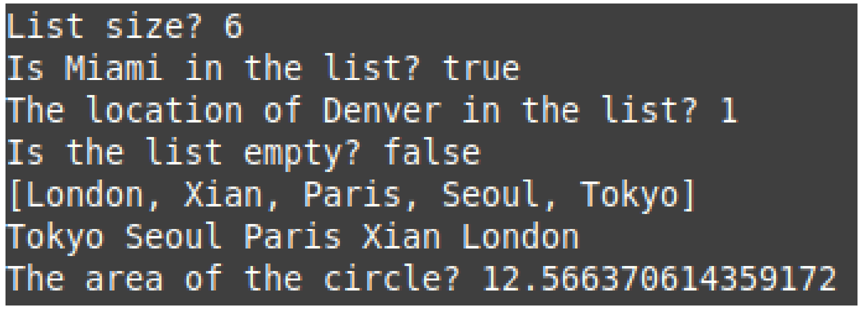
- ArrayList<String> cities = **new** ArrayList<>();

Array vs ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

```java
import java.util.ArrayList;

public class TestArrayList {
  public static void main(String[] args) {
    ArrayList<String> cityList = new ArrayList<>();

    cityList.add("London");
    cityList.add("Denver");
    cityList.add("Paris");
    cityList.add("Miami");
    cityList.add("Seoul");
    cityList.add("Tokyo");

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " + cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? " + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " + cityList.isEmpty());

    cityList.add(2, "Xian");
    cityList.remove("Miami");
    cityList.remove(1);
    System.out.println(cityList.toString());

    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    ArrayList<Circle> list = new java.util.ArrayList<>();

    list.add(new Circle(2));
    list.add(new Circle(3));

    System.out.println("The area of the circle? " +
      list.get(0).getArea());
  }
}
```

```java
import java.util.ArrayList;

public class TestArrayList {
  public static void main(String[] args) {
    ArrayList<String> cityList = new ArrayList<>();

    cityList.add("London");
    cityList.add("Denver");
    cityList.add("Paris");
    cityList.add("Miami");
    cityList.add("Seoul");
    cityList.add("Tokyo");

    System.out.println("List size? " + cityList.size());
    System.out.println("Is Miami in the list? " + cityList.contains("Miami"));
    System.out.println("The location of Denver in the list? " + cityList.indexOf("Denver"));
    System.out.println("Is the list empty? " + cityList.isEmpty());

    cityList.add(2, "Xian");
    cityList.remove("Miami");
    cityList.remove(1);
    System.out.println(cityList.toString());

    for (int i = cityList.size() - 1; i >= 0; i--)
      System.out.print(cityList.get(i) + " ");
    System.out.println();

    ArrayList<Circle> list = new java.util.ArrayList<>();

    list.add(new Circle(2));
    list.add(new Circle(3));

    System.out.println("The area of the circle? " +
      list.get(0).getArea());
  }
}
```



```
List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
Tokyo Seoul Paris Xian London
The area of the circle? 12.566370614359172
```

Naming convention

- When using **generics** in Java, the **type parameters** are typically represented by **single uppercase letters**. This is in contrast to regular variable names.
- Without this convention, it would be difficult to tell the difference between a **type variable** and an **ordinary class** or **interface name.**

Naming convention

- Commonly used type parameters:

Type parameter	Meaning
E	Element, used in collections, e.g., List<E>.
K	Key, used in maps, e.g., Map<K,V>
V	Value, used in maps, e.g., Map<K,V>
N	Number, represents numeric type.
T	Type, general placeholder for any type.
S, U, V, etc.	Used when multiple type parameters are needed.

Generic types

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

list.add(“Red”);   //ok

list.add(~~new Integer~~(1));

List is already defined as **ArrayList of String**. If you attempt to add a **nonstring**, a **compile error** will occur.

Generic types

```
ArrayList<Integer> x = new ArrayList<>();  //OK
ArrayList<Double> y = new ArrayList<>();  //OK
```

$\cancel{\text{ArrayList<int> x = new ArrayList<>();}}$

E must be **reference types** (e.g. Integer, Double, MyPet), cannot be a **primitive type** (e.g. int, double, char etc.)

No Casting Needed

- The following code snippet without generics requires casting:

```
ArrayList list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
//Casting needed prior to JDK1.5 or you will get compile error - incompatible types.
```

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

```
ArrayList<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0); // no casting is needed
```

Check Point 1

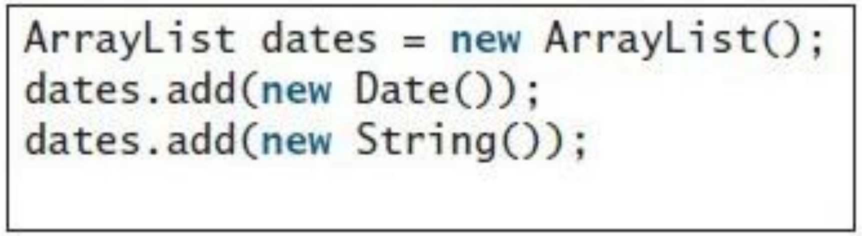
1. What is the benefit of using generic types?

Check Point 1

1. What is the benefit of using generic types?
   - » Stronger type checks at compile time, improving reliability.
   - » Elimination of casting.

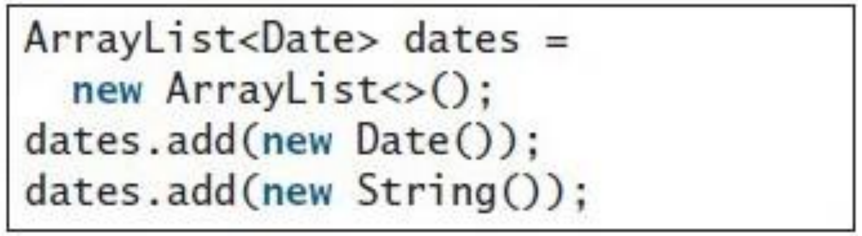
Check Point 1

2. Are there any compile errors in (a) and (b)?



```
ArrayList dates = new ArrayList();
dates.add(new Date());
dates.add(new String());
```

(a) Prior to JDK 1.5

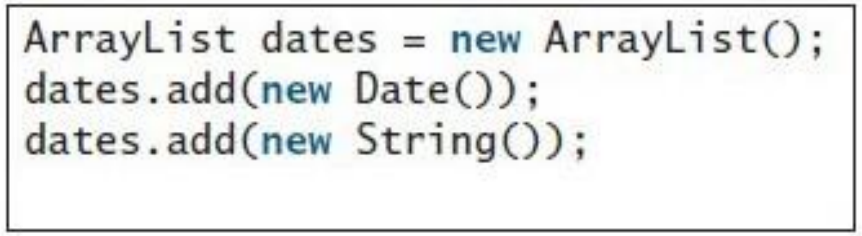


```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
dates.add(new String());
```

(b) Since JDK 1.5

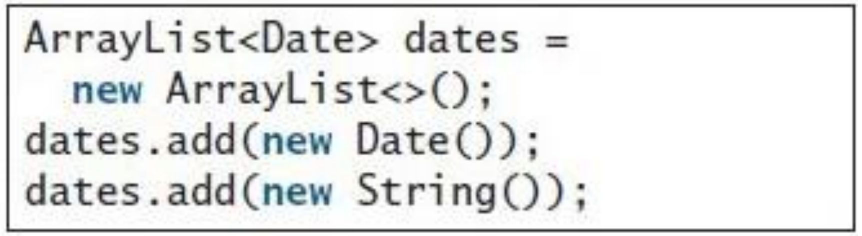
Check Point 1

2. Are there any compile errors in (a) and (b)?



```
ArrayList dates = new ArrayList();
dates.add(new Date());
dates.add(new String());
```

(a) Prior to JDK 1.5



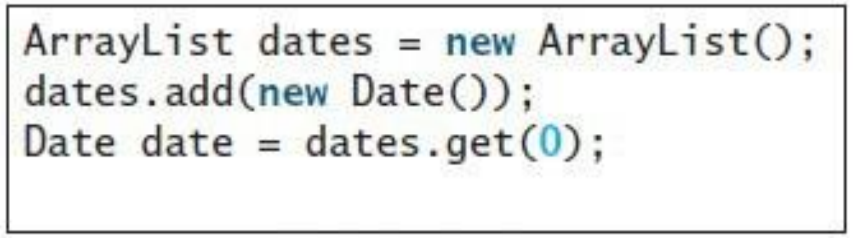
```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
dates.add(new String());
```

(b) Since JDK 1.5

(a) No compile error
(b) Compilation error – argument mismatch.
String cannot be converted to **Date**

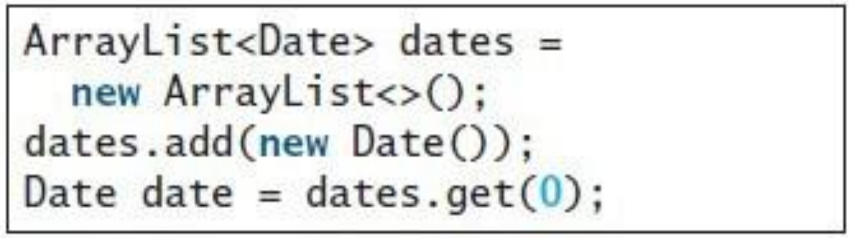
Check Point 1

3. What is wrong in (a)? Is code in (b) correct?



```
ArrayList dates = new ArrayList();
dates.add(new Date());
Date date = dates.get(0);
```

(a) Prior to JDK 1.5

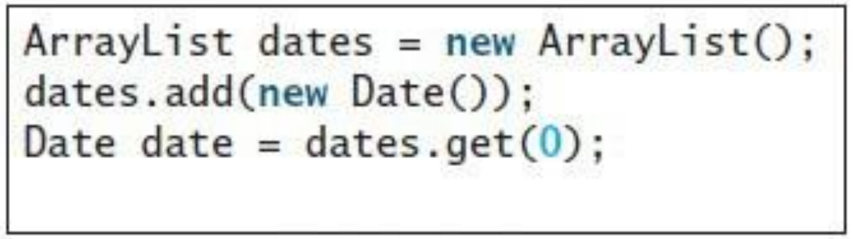


```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
Date date = dates.get(0);
```

(b) Since JDK 1.5

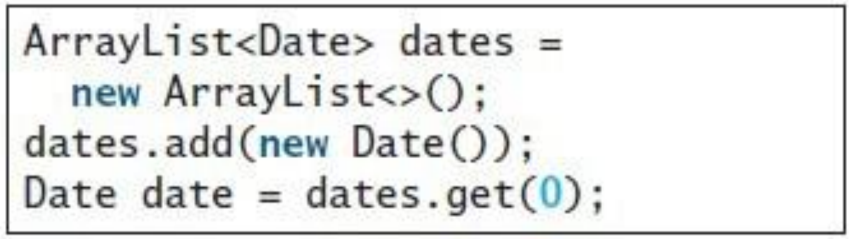
Check Point 1

3. What is wrong in (a)? Is code in (b) correct?



```
ArrayList dates = new ArrayList();
dates.add(new Date());
Date date = dates.get(0);
```

(a) Prior to JDK 1.5



```
ArrayList<Date> dates =
  new ArrayList<>();
dates.add(new Date());
Date date = dates.get(0);
```

(b) Since JDK 1.5

(a) Need casting – Object cannot be converted to Date

Date date =(Date)dates.get(0);

(b) No casting is needed

Generic types

Generics can be defined for a :

1. Class
2. Interface
3. Method

Declaring Generic Class

Syntax:

```
public class ClassName<E> { }
```

e.g.

```
public class GenericStack<E> {……}
```

Declaring Generic Interface

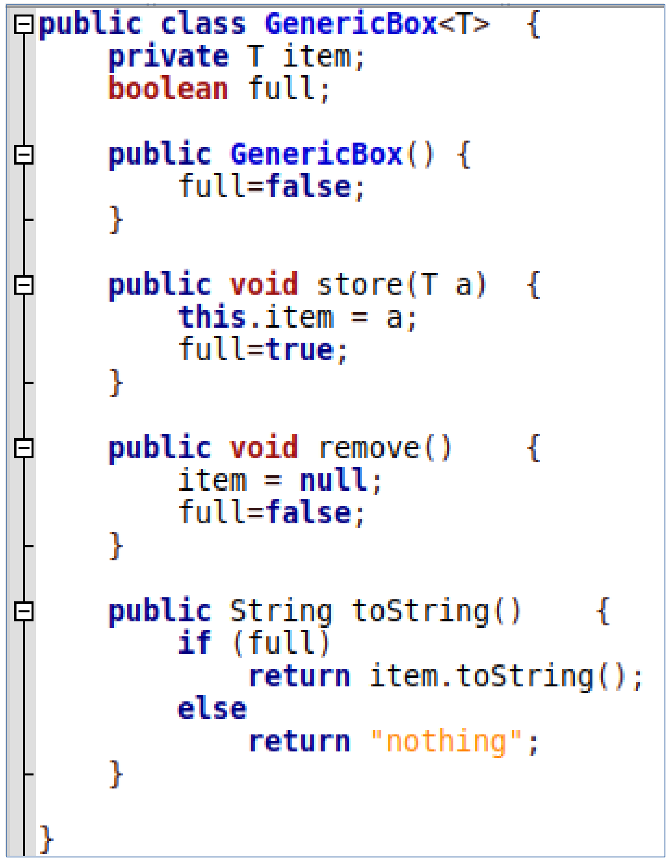
Syntax:

```
  public class InterfaceName<E> { }
```

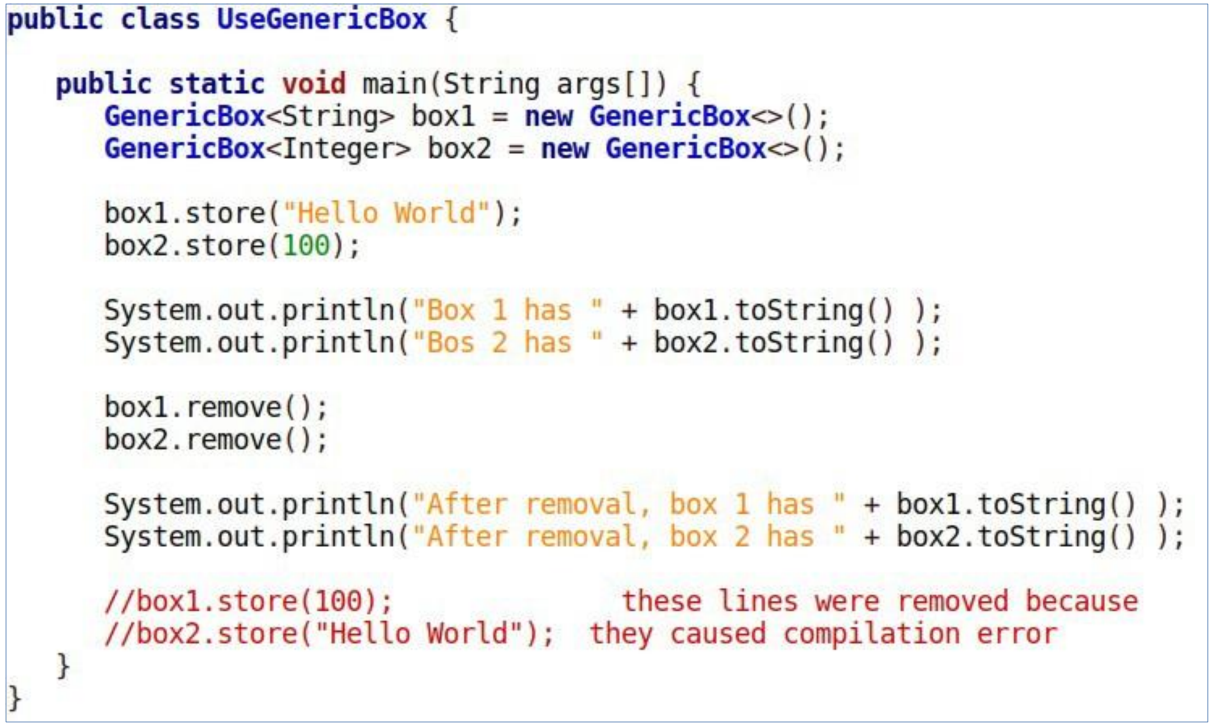
e.g.

```
  public interface Comparable<E> {......}
  public interface Edible<E> {......}
```

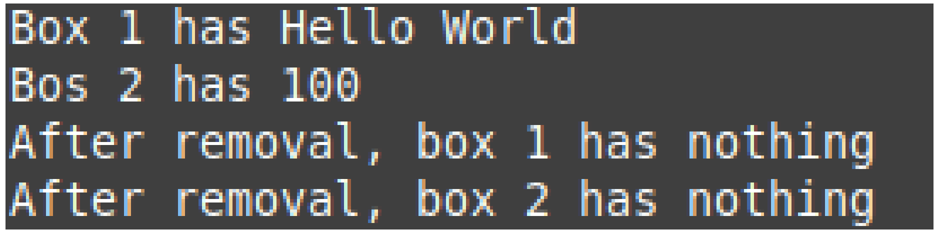
Example: GenericBox



```java
public class GenericBox<T>  {
    private T item;
    boolean full;

    public GenericBox() {
        full=false;
    }

    public void store(T a)  {
        this.item = a;
        full=true;
    }

    public void remove()    {
        item = null;
        full=false;
    }

    public String toString()    {
        if (full)
            return item.toString();
        else
            return "nothing";
    }

}
```



```java
public class UseGenericBox {

   public static void main(String args[]) {
      GenericBox<String> box1 = new GenericBox<>();
      GenericBox<Integer> box2 = new GenericBox<>();

      box1.store("Hello World");
      box2.store(100);

      System.out.println("Box 1 has " + box1.toString() );
      System.out.println("Bos 2 has " + box2.toString() );

      box1.remove();
      box2.remove();

      System.out.println("After removal, box 1 has " + box1.toString() );
      System.out.println("After removal, box 2 has " + box2.toString() );

      //box1.store(100);            these lines were removed because
      //box2.store("Hello World");  they caused compilation error
   }
}
```



```
Box 1 has Hello World
Bos 2 has 100
After removal, box 1 has nothing
After removal, box 2 has nothing
```

Points to Take Note

1. Generics class constructor is defined as

```
public GenericStack()
```

2. Generic class may have more than 1 parameter.

```
<E1, E2, E3>
```

3. You can define a class/interface as a subtype of a generic class/interface. For example,

```
public class String implements Comparable<String>
//where String class implements Comparable interface
```

Check Point 2

1. Declare generic class and generic interface.

Check Point 2

1. Declare generic class and generic interface.

```
public class ClassName<E>{…}
public interface InterfaceName<E>{…}
```

Check Point 2

2. Define generic class and interface.

Check Point 2

2. Define generic class and interface.

   A **generic type** is a **class or interface** that uses a **type parameter** (T, E, K, etc.), which is parameterized over **type**, such as (String, Integer, etc.).

Check Point 2

3. Can a generic class have multiple generic parameters?

Check Point 2

3. Can a generic class have multiple generic parameters?

   Yes, `<E1,  E2, E3>`

Generic Methods

Syntax :

```
public static <E> returnType methodName(E parameter)
```

Example :

```
public static <E> void print(E[] list)
public <E> boolean isFilled(E filled)
```

Generic Methods

To declare generics:

- place <E> before returnType

To invoke/call method:

- place <actualType>methodName
- or as usual method call

LISTING 19.2 GenericMethodDemo.java

```java
public class GenericMethodDemo {
  public static void main(String[] args ) {
    Integer[] integers = {1, 2, 3, 4, 5};
    String[] strings = {"London", "Paris", "New York", "Austin"};

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

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

generic method

Bounded Generic Type

Bounded generic type is a generic type specified as a **subtype of another type**, such as,

`<E extends GeometricObject>`

`E` is a **generic type** of `GeometricObject`.

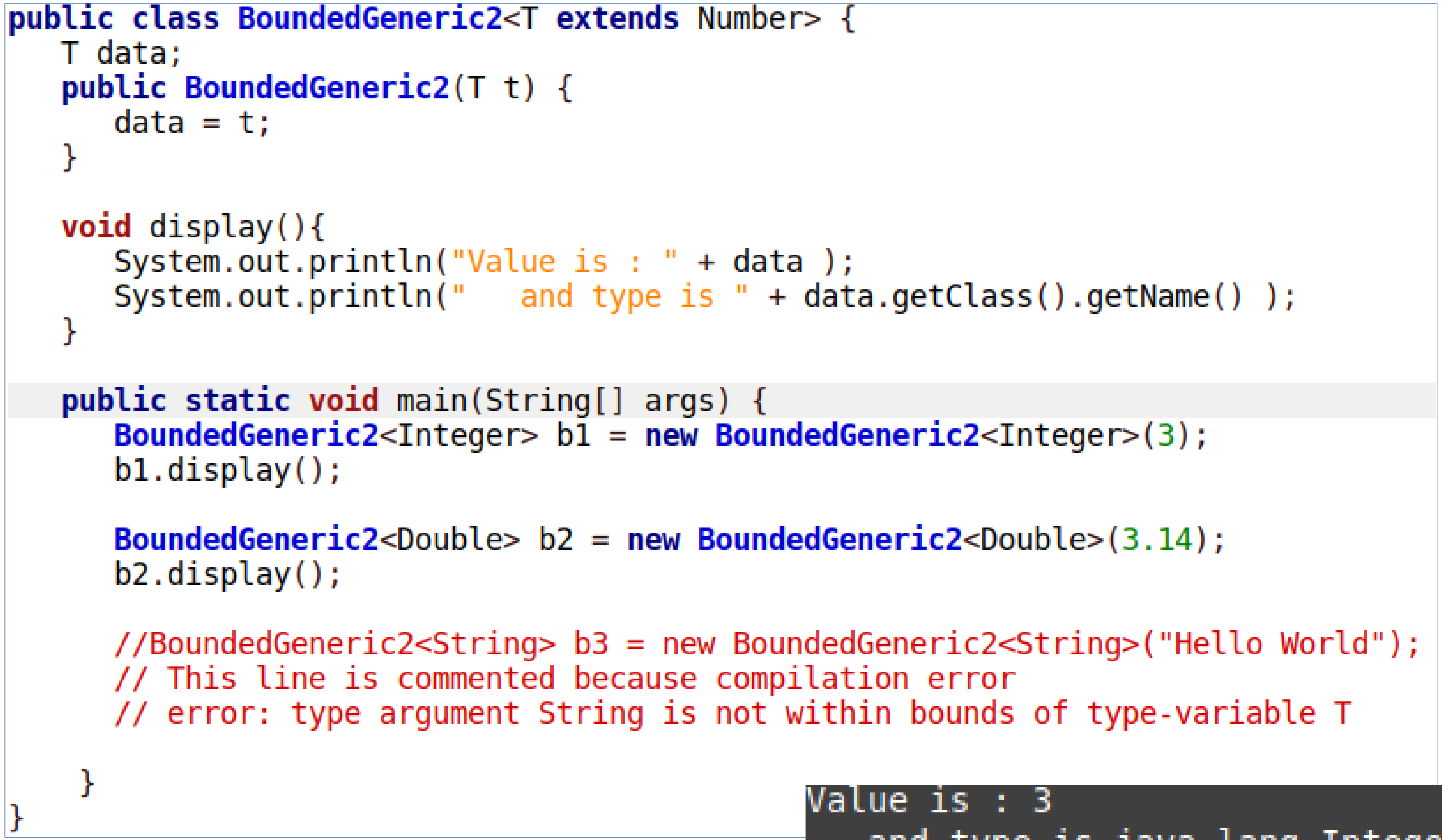
An unbounded generic type <E> is the same as,

`<E extends Object>`

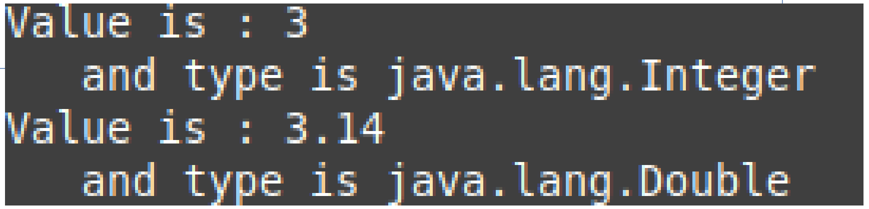
Bounded Type Parameter

- Sometimes, you may want to **limit the types** that can be used as arguments in a **generic type**.
- For example, if a method should only work with numbers, you can **restrict** it to accept only Number or its subclasses (e.g., Integer, Double). This is done using **bounded type parameters**.
- To create a bounded type parameter, use the **extends** keyword followed by the upper bound (Number in this case).

```
public <U extends Number> void inspect(U u)
```



```java
public class BoundedGeneric2<T extends Number> {
   T data;
   public BoundedGeneric2(T t) {
      data = t;
   }

   void display(){
      System.out.println("Value is : " + data );
      System.out.println("   and type is " + data.getClass().getName() );
   }

   public static void main(String[] args) {
      BoundedGeneric2<Integer> b1 = new BoundedGeneric2<Integer>(3);
      b1.display();

      BoundedGeneric2<Double> b2 = new BoundedGeneric2<Double>(3.14);
      b2.display();

      //BoundedGeneric2<String> b3 = new BoundedGeneric2<String>("Hello World");
      // This line is commented because compilation error
      // error: type argument String is not within bounds of type-variable T

    }
}
```



```
Value is : 3
   and type is java.lang.Integer
Value is : 3.14
   and type is java.lang.Double
```

Check Point 3

1. Declare a static generic method for `myMethod` that does not return anything, but accepts one parameter called `a`.
2. What is bounded generic type?

Check Point 3

1. Declare a static generic method for `myMethod` that does not return anything, but accepts one parameter called `a`.

   ```
   public static <E> void myMethod (E a)
   ```

2. What is bounded generic type?

   Generic type specified as a subtype of another type.

Raw Type and Backward Compatibility

- **Raw type** : a generic class/interface used **without specifying a concrete type/without a type parameter.**

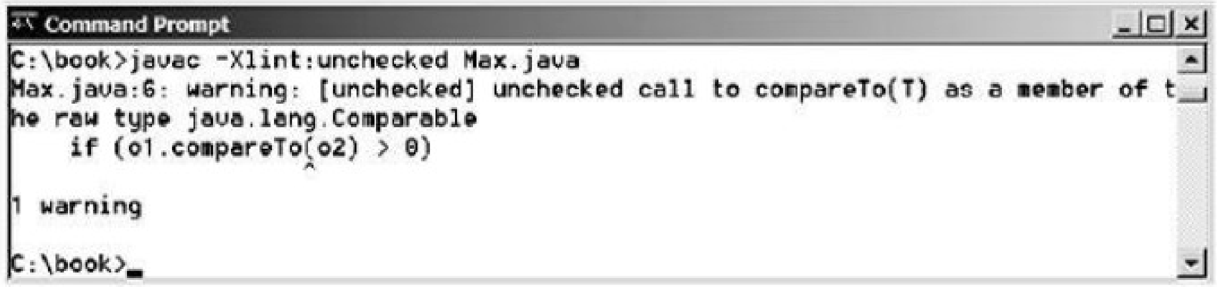
```
// raw type
ArrayList list = new ArrayList();
```

- This is *roughly* equivalent to

```
ArrayList<Object> list = new ArrayList<Object>();
```

Raw Type is Unsafe

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum between two objects */
  public static Comparable max(Comparable o1,Comparable o2) {
    if (o1.compareTo(o2) > 0) return o1;
    else return o2;
  }
}
```



```
C:\book>javac -Xlint:unchecked Max.java
Max.java:6: warning: [unchecked] unchecked call to compareTo(T) as a member of t
he raw type java.lang.Comparable
    if (o1.compareTo(o2) > 0)
                    ^
1 warning

C:\book>_
```

```
Runtime Error:
Max.max("Welcome", 23);
```

Note: **Comparable o1** and **Comparable o2** are *raw type declarations*. 23 is autoboxed into new Integer(23)

Make it Safe (by using Generic type)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum between two objects */
  public static <E extends Comparable<E>> E max(E o1 , E o2) {
    if (o1.compareTo(o2) > 0) return o1;
    else return o2;
  }
}
```

```
Max.max("Welcome", 23); //will be compile error
```

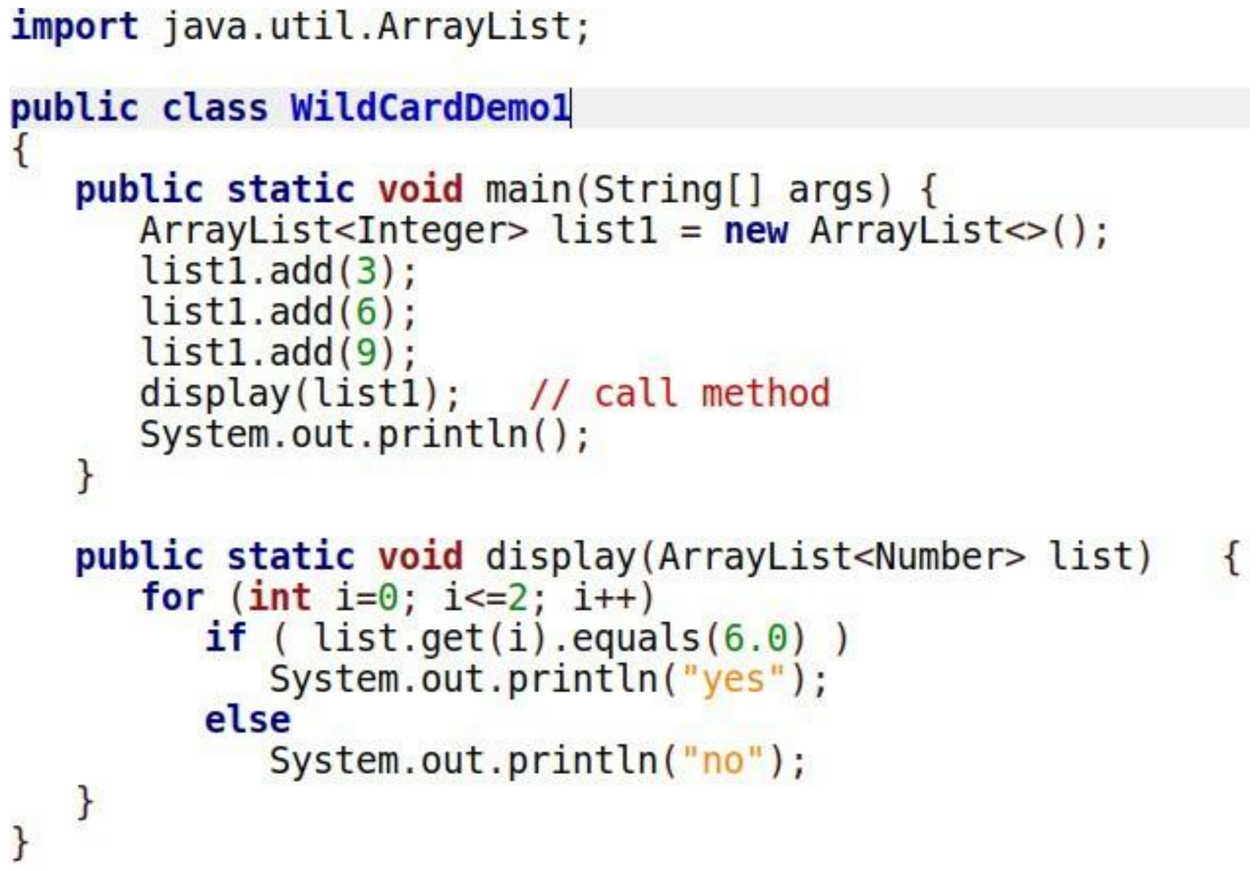
Avoiding Unsafe Raw Types

✓**Use** : `new ArrayList<ConcreteType>();`

x Instead of: `new ArrayList();`

Wildcard Generic Types

- Why wildcards are necessary? See this example.

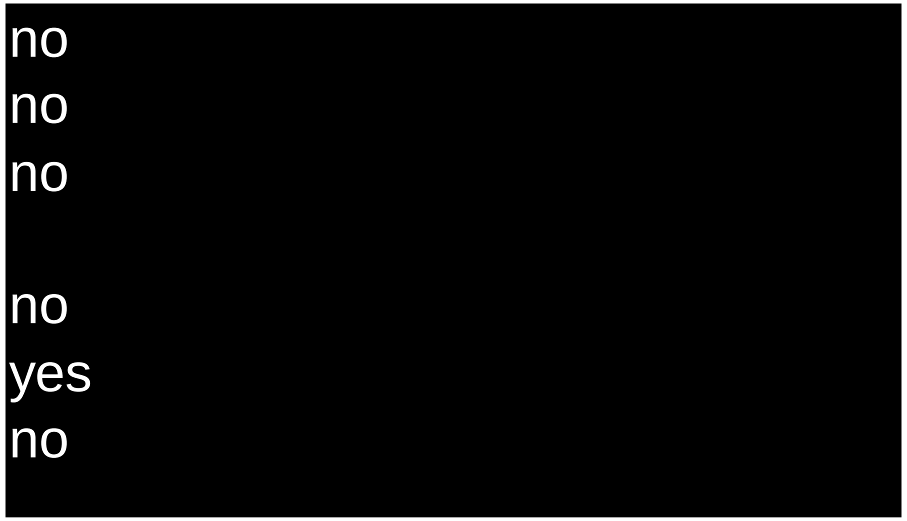


```
import java.util.ArrayList;

public class WildCardDemo1
{
    public static void main(String[] args) {
        ArrayList<Integer> list1 = new ArrayList<>();
        list1.add(3);
        list1.add(6);
        list1.add(9);
        display(list1);    // call method
        System.out.println();
    }

    public static void display(ArrayList<Number> list)    {
        for (int i=0; i<=2; i++)
            if ( list.get(i).equals(6.0) )
                System.out.println("yes");
            else
                System.out.println("no");
    }
}
```

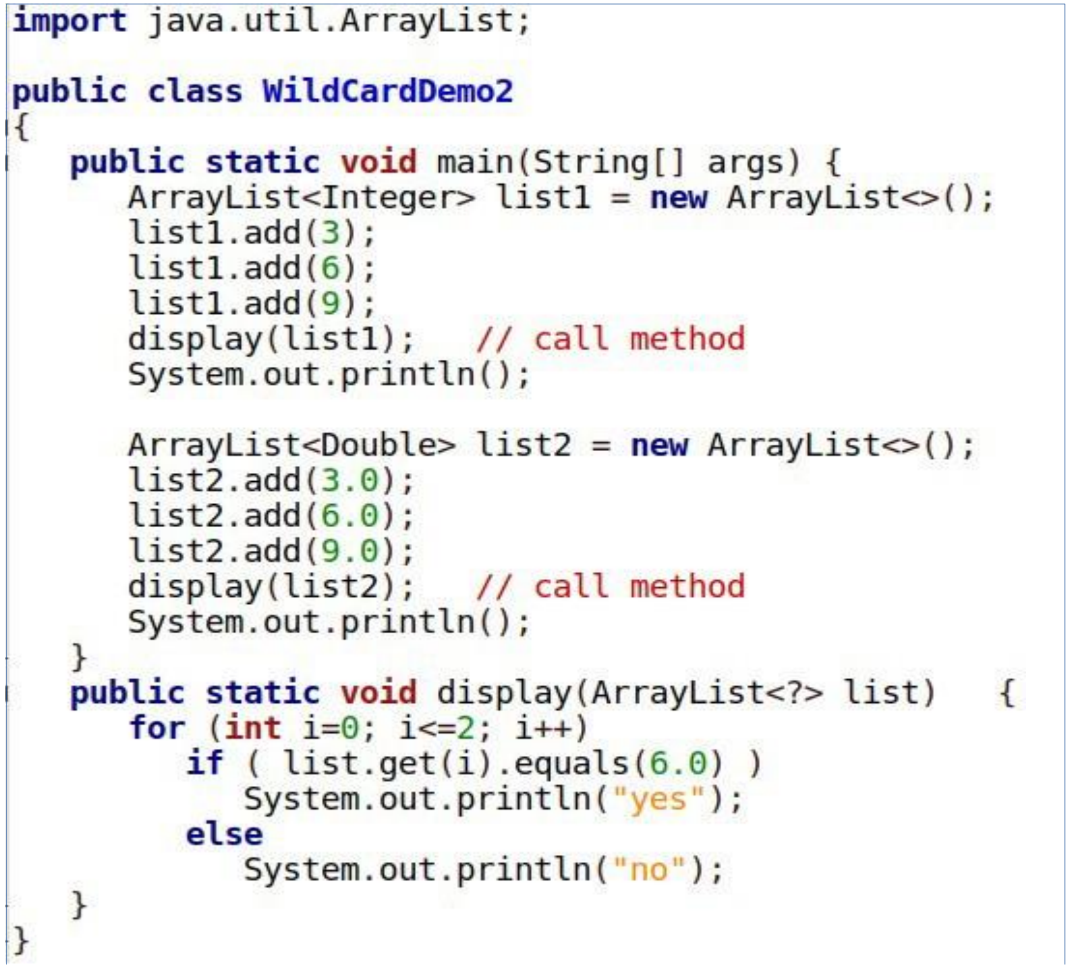
//compile error:
Integer is a subtype of Number.
However, list1, which is an instance of ArrayList<Integer>, is not an instance of ArrayList<Number>

Wildcard Generic Types

- Wildcards (?) represents an **"unknown type"**



```
no
no
no

no
yes
no
```



```java
import java.util.ArrayList;

public class WildCardDemo2
{
   public static void main(String[] args) {
      ArrayList<Integer> list1 = new ArrayList<>();
      list1.add(3);
      list1.add(6);
      list1.add(9);
      display(list1);    // call method
      System.out.println();

      ArrayList<Double> list2 = new ArrayList<>();
      list2.add(3.0);
      list2.add(6.0);
      list2.add(9.0);
      display(list2);    // call method
      System.out.println();
   }
   public static void display(ArrayList<?> list)   {
      for (int i=0; i<=2; i++)
         if ( list.get(i).equals(6.0) )
            System.out.println("yes");
         else
            System.out.println("no");
   }
}
```

Wilcard Generic Types

3 forms of wildcard generic type:

?	unbounded wildcard (represents any object type)
? extends T	bounded wildcard (unknown subtype of T)
? super T	lower bound wildcard (unknown supertype of T)

- <?> equals <? extends Object>

Wilcard Generic Types

- 3 forms of wildcard generic type:

(1) ? = unbounded wildcard (represents any object type)

(2) ? extends T = bounded wildcard (unknown subtype of T)

(3) ? super T = lower bound wildcard (unknown supertype of T)

Syntax:

```
<?> equals <? extends Object>
```

Erasure and Restrictions on Generics

- Generics are implemented using an approach called *type erasure*. The **compiler uses the generic type information to compile the code, but erases it afterwards**.
- Hence, the generic information is **not available at run time**. This approach enables the generic code to be ***backward-compatible*** with the legacy code that uses **raw types**.

Compile Time Checking

For example, the compiler checks whether generics is used correctly for the following code in (a) and translates it into the equivalent code in (b) for runtime use. The code in (b) uses the raw type.

```
ArrayList<String> list = new ArrayList<String>();
list.add("Oklahoma");
String state = list.get(0);
```

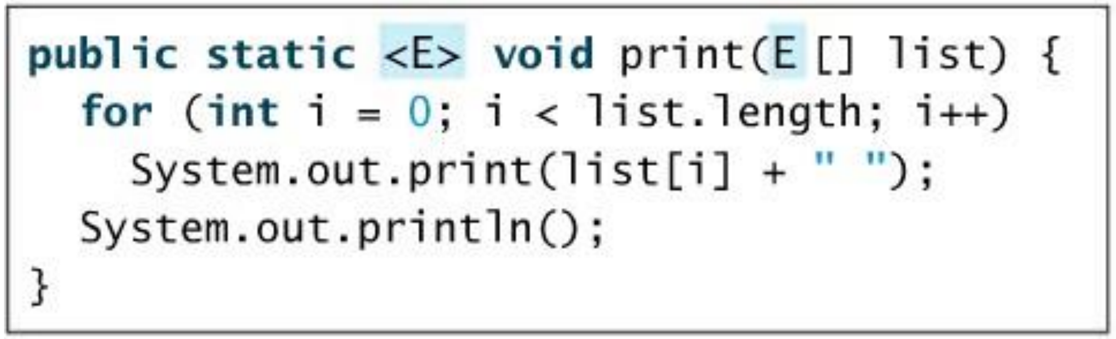
(a)

```
ArrayList list = new ArrayList();
list.add("Oklahoma");
String state = (String)(list.get(0));
```

(b)

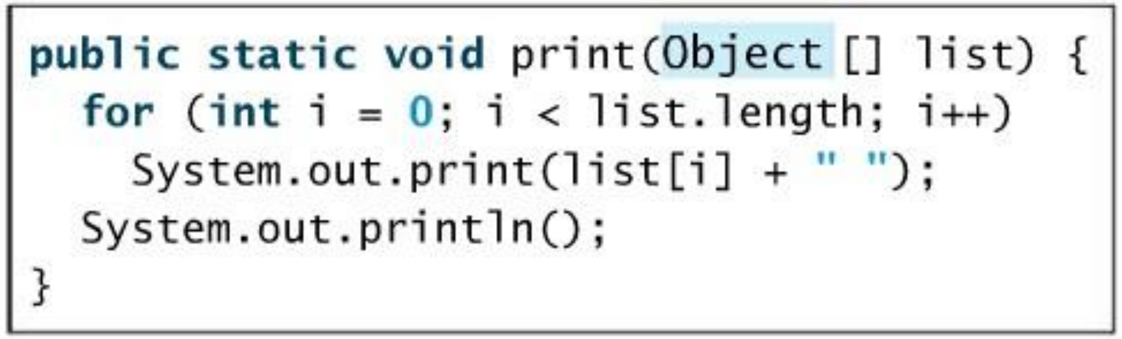
Compile Time Checking

When generic classes, interfaces and methods are compiled, the compiler replaces the generic type with the **Object** type.



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

(a)

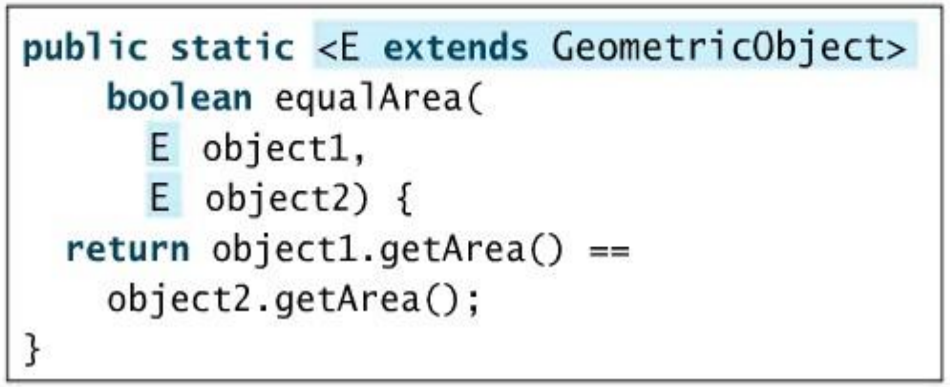


```
public static void print(Object [] list) {
  for (int i = 0; i < list.length; i++)
    System.out.print(list[i] + " ");
  System.out.println();
}
```

(b)

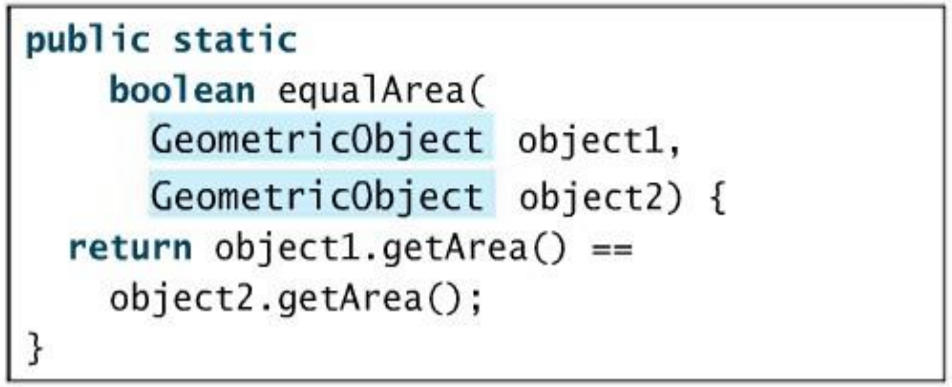
Compile Time Checking

If a generic type is bounded, the compiler replaces it with the bounded type.



```
public static <E extends GeometricObject>
    boolean equalArea(
      E object1,
      E object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(a)



```
public static
    boolean equalArea(
      GeometricObject object1,
      GeometricObject object2) {
  return object1.getArea() ==
    object2.getArea();
}
```

(b)

Important Facts

Note that a **generic class is shared by all its instances** regardless of its actual generic type.

```
ArrayList<String> list1 = new ArrayList<String>();
ArrayList<Integer> list2 = new ArrayList<Integer>();
System.out.println(list1 instanceof    ArrayList); //true
System.out.println(list2 instanceof    ArrayList); //true
```

Although `ArrayList<String>` and `ArrayList<Integer>` are two types, but there is only one class `ArrayList` loaded into the JVM.

Restrictions on Generic

1. CAN NOT create an instance of a generic type. (i.e., new E()).
2. Generic array creation IS NOT ALLOWED. (i.e., new E[100]).
3. A generic type parameter of a class IS NOT ALLOWED in a static context.
4. Exception classes CAN NOT be Generic.

Restriction 1 : CAN NOT use new E()

```
E object = new E();
```

new E() is executed at runtime, but E not available

Restriction 2 : CAN NOT use new E[]

```
E[] elements = new E[capacity]
```

Avoid error by having :

```
E[] elements = (E[]) new Object[capacity]
```

But causes **unchecked compile warning** because compiler not certain that casting will succeed at runtime. If E is String, new Object[] is an array of Integer objects, then ((String[]) new Object[]) will cause a compile error (*ClassCastException*)

Restriction 3 : NOT ALLOWED in Static Context

```
public class Test<E> {
    public static void m(E o1) { //Illegal static method
        //some codes
    }
    public static E o1; //Illegal field

    static {
        E o2;  //Illegal
    }
}
```

Restriction 4 : Exception class CAN NOT be Generic

- Generic class may not extend java.lang.Throwable. So, the following is illegal,

```
public class MyException<T> extends Exception { }
```

- Because the type information is not present at runtime.

```
try {
   ...
}
catch(MyException<T> ex) {
   ...
}
```

Check Point 4

1. If a program uses `ArrayList<String>` and `ArrayList<Date>`, does the JVM load both of them?
2. What is the problem with this code :

```
E object = new E();
```

3. Can you define a generic exception class? Why?

Check Point 4

1. If a program uses `ArrayList<String>` and `ArrayList<Date>`, does the JVM load both of them?
   1 class loaded into the JVM.

2. What is the problem with this code :

   ```
   E object = new E();
   ```

   new E() executed at runtime but E not available.

Check Point 4

3. Can you define a generic exception class? Why?

   CAN NOT. Because the type information is not present at runtime.

References

1. Chapter 19 Generics, Liang, Introduction to Java Programming, 10th Edition, Global Edition, Pearson, 2015.
2. https://docs.oracle.com/javase/tutorial/java/generics/types.html