Generics

Shing-Chi Cheung
Computer Science and Engineering
HKUST

Motivations

- Only raw type was supported before Java 5

```
List cities = new ArrayList();
cities.add(new Object());
cities.add("Hong Kong");
cities.add(Integer.valueOf(1));
```

```
var cities = new ArrayList<String>();
cities.add(new Object());          ✘
cities.add("Hong Kong");           ✓
cities.add(Integer.valueOf(1));    ✘
```

- 👎 Raw type allows any type to be added
- 👎 Lead to potential casting exceptions at runtime

Motivations

- Generics are widely used in Java
  - Examples: Comparable<E> and ArrayList<E>

type parameter

- Generics let us define a class, an interface or a method with type parameters/variables (e.g., E) to be substituted with actual type arguments (e.g., String) at compilation
  - **var** cities = **new** ArrayList<String>(); *// ArrayList of String*
  - ArrayList<String> cities = **new** ArrayList<>();

diamond operator introduced in Java 7 to infer the appropriate type used

Terminology

```
class ArrayList<E> {
  …
}
```

```
void func {
  var cities = new ArrayList<String>();
  …
}
```

- ArrayList<E> is a generic class or generic type
- E is a type parameter or type variable
- ArrayList<String> is a parameterized type of ArrayList<E>
- String is an actual type argument for E

Syntax

- Generics allow us to parameterize a class, an interface and a method with type parameters
- Actual type arguments must be object reference types (e.g., Object, String, Circle, Comparable)

Syntax:

<access modifier> class <class name> <<type par>> { … }

<access modifier> interface <interface name> <<type par>> { … }

<access modifier> <<type par>> <return type> <method name>
(<type par> <param name>, …) { … }

Usage

- For example, we may define a generic stack class that stores the elements of any type.

```
public class GenericStack<E> { … }  // define a generic stack of type E
```

- We may create a stack object to hold strings and a stack object to hold numbers. String and Number are actual type arguments for the type parameter E.

```
var stackOfString = new GenericStack<String>();
var stackOfInteger = new GenericStack<Number>();
```

Usage

```java
public class GenericStack<E> {
  private ArrayList<E> list = new ArrayList<>();
  public int getSize() { return list.size(); }
  public E peek() { return list.get(getSize() - 1); }
  public void push(E o) { list.add(o); }
  public E pop() {
    E o = list.get(getSize() - 1);    // Before Java 10
    list.remove(getSize() - 1);
    return o;
  }
  public boolean isEmpty() { return list.isEmpty(); }
  @Override
  public String toString() { return "stack: " + list.toString(); }
}
```

GenericStack.java

Usage

No need to put down E o = list…

```
public class GenericStack<E> {
  private ArrayList<E> list = new ArrayList<>();
  public int getSize() { return list.size(); }
  public E peek() { return list.get(getSize() - 1); }
  public void push(E o) { list.add(o); }
  public E pop() {
    var o = list.get(getSize() - 1);// since Java 10
    list.remove(getSize() - 1);
    return o;
  }
  public boolean isEmpty() { return list.isEmpty(); }
  @Override
  public String toString() { return "stack: " + list.toString(); }
}
```

With the support of local variable type inference, we do not need to use type parameters to declare local variables

➔ more maintainable code

Why Generics?

- A key benefit of generics is to enable errors to be detected at compile time rather than at runtime.
- A generic class or method enables us to specify eligible types of arguments that the class or method may work with.
- If we attempt to use the class or method with an incompatible argument, a compile error occurs.

Generic Type

Compare.java

```
package java.lang;

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

(a) Prior to JDK 1.5

```
package java.lang;

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

(b) After JDK 1.5

Generic Instantiation

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) After JDK 1.5

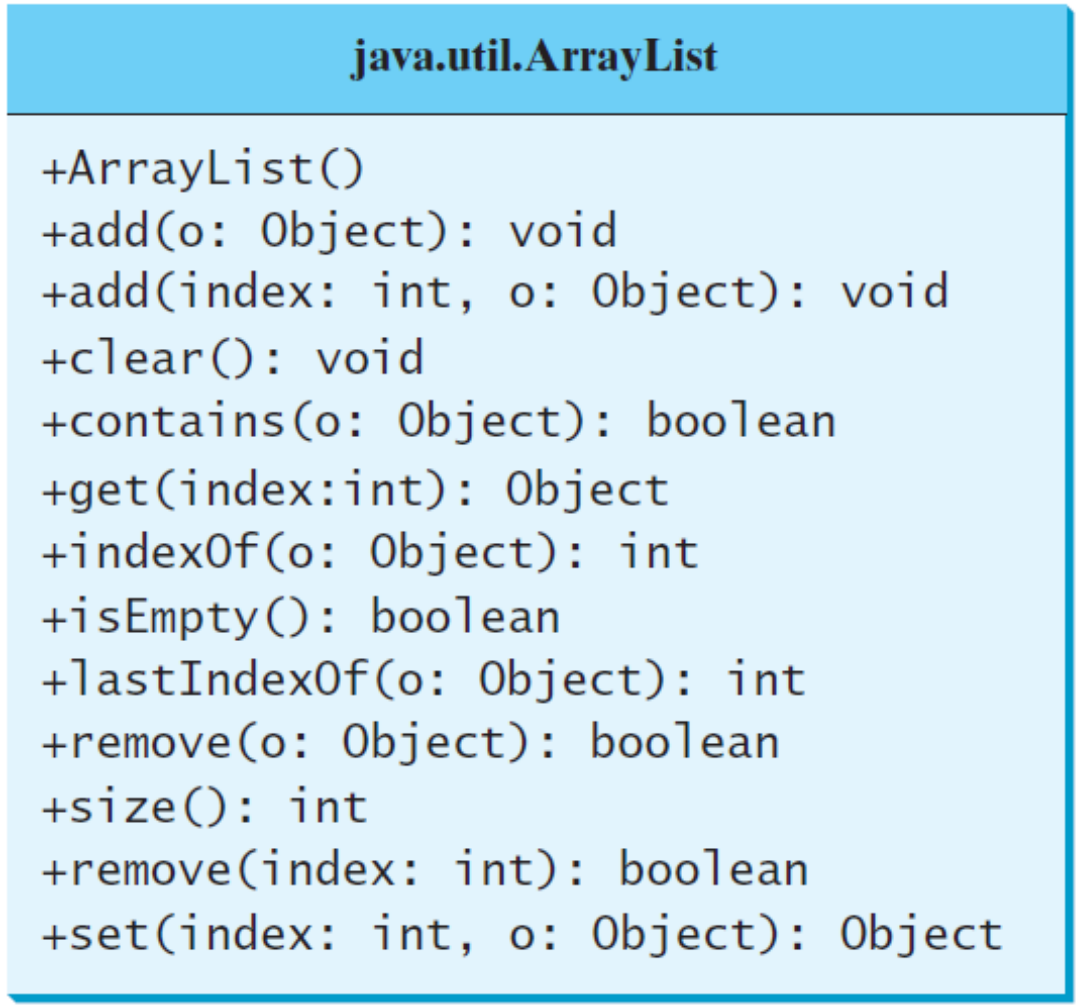
Compile error

Detecting errors at compile time improves reliability

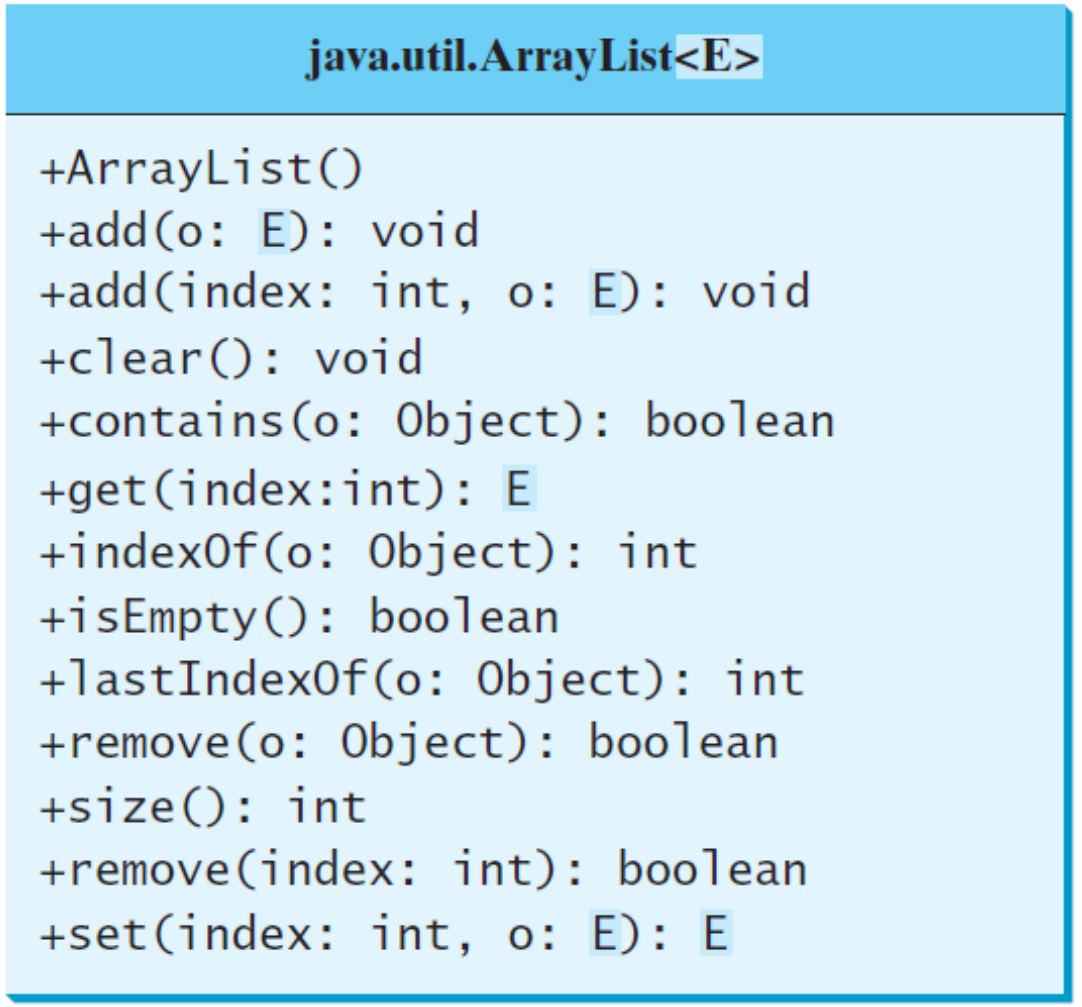
Why Generics?

- A key benefit of generics is to enable errors to be detected at compile time rather than at runtime.
- A generic class or method enables us to specify eligible types of arguments that the class or method may work with.
- If we attempt to use the class or method with an incompatible argument, a compilation error occurs.

Generic ArrayList after JDK 1.5



(a) ArrayList before JDK 1.5



(b) ArrayList since JDK 1.5

No Casting Needed if Type Parameter is binded

- Casting is not needed to retrieve a value from an arraylist created with an actual argument type (say String)
- The String type of s will be inferred

```
var list = new ArrayList<String>();
list.add("Red");
list.add("White");
var s = list.get(0); // No casting is needed
```

```
var list = new ArrayList();
list.add("Red");
list.add(1);
String s = (String) list.get(0); // Casting is needed
```

No Casting is Needed for Wrapper Types

- If the type of an arraylist's element is a wrapper type (e.g., Short, Integer, Double and Character) we can assign the element to its corresponding primitive type variable (known as auto-unboxing)

```
var list = new ArrayList<Double>(); // is incompatible with ArrayList<Integer>
list.add(5.5); // double value 5.5 is auto-boxed to new Double(5.5)
//   list.add(3); //  3 cannot be automatically converted to new Double(3.0)
list.add((double) 3);
var doubleObject = list.get(0); // No casting is needed
double d = list.get(1); // auto-unboxing a Double object to a double value
```

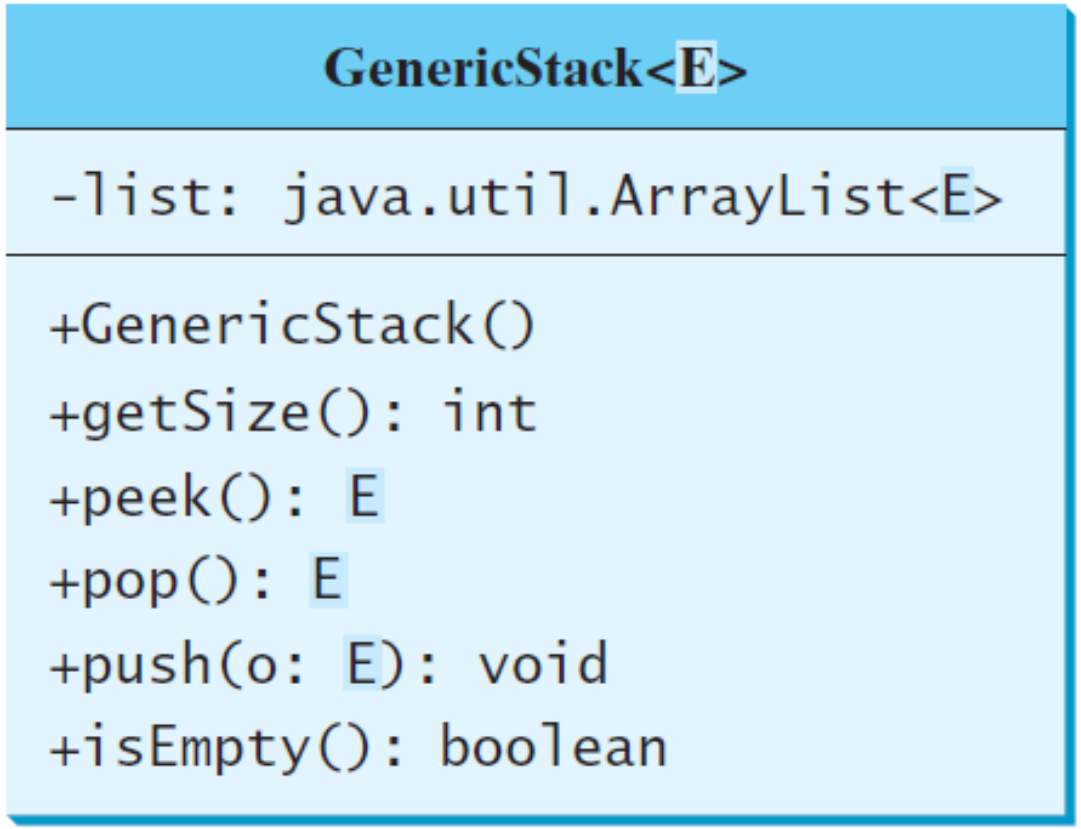
Casting.java

Comparison between Array and 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); or list[index];
Updating an element	a[index] = "London";	list.set(index, "London"); or list[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);
Updating an element		list.remove(Object);

Some operations are not friendly to type propagation:
Even A is a super type of B, ArrayList<A> is not a super type of ArrayList<B>.

Declaring Generic Classes and Interfaces



An array list to store elements.	
Creates an empty stack.	
Returns the number of elements in this stack.	
Returns the top element in this stack.	
Returns and removes the top element in this stack.	
Adds a new element to the top of this stack.	
Returns true if the stack is empty.	

GenericStack.java

Two Implementations of Generic Methods

In front of the return type, declare type parameter before use

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

(a) Use Generic Type

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

(b) Use Object

Advantage: Allows printing an inappropriate list to trigger a compilation error

GenericMethodDemo.java

Raw Type Supported for Backward Compatibility

Raw type refers to the use of a generic class, interface or method without assigning actual type arguments

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

The effect is ***roughly*** similar to:

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

Or more accurately it can be thought of:

```
var list = new ArrayList<?>(); // wildcard type ? will be discussed
```

Avoiding Unsafe Raw Types

Raw types are unsafe as we will show in the next slide

So use

```
new ArrayList<ConcreteType>()
```

instead of

```
new ArrayList();
```

TestArrayListNew.java

Raw Type is Unsafe - 1

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

```
public class ComparableCircleWithoutGeneric extends Circle
    implements Comparable {
    …
public int compareTo(Object o) {
    double diff = this.getArea() - ((Circle) o).getArea();
    if (diff > 0)
      return 1;
    else if (diff < 0)
      return -1;
    else
      return 0;
  }
}
```

```
// triggers a runtime error if o is not passed a circle
new ComparableCircleWithoutGeneric(1).compareTo("RED");
```

ComparableCircleWithoutGeneric.java

Make it Safe with Generics

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

```
public class ComparableCircleWithGeneric extends Circle
    implements Comparable<Circle> {
    …
  public int compareTo(Circle o) {
    double diff = this.getArea() - o.getArea(); // casting is not needed
    if (diff > 0)
      return 1;
    else if (diff < 0)
      return -1;
    else
      return 0;
  }
}
```

```
// triggers a compilation error if o is not passed a circle
new ComparableCircleWithGeneric(1).compareTo("RED");
```

ComparableCircleWithGeneric.java

Raw Type is Unsafe - 2

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

```
public class Max {
  public static Comparable max(Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

- Max.max("Welcome", 23); *// runtime error!*
  - *String cannot be compared with an integer*

Max.java

Raw Type is Unsafe - 2

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

```
public class Max {
  public static Comparable max(Comparable o1, Comparable o2) {
```

```
public static <E> Comparable max(E o1, E o2) { // ?
```

```
public static <E extends Comparable> E max(E o1, E o2) { // ?
```

bounded type parameter

```
public static <E extends Comparable<E>> E max(E o1, E o2) { // ?
```

- Max.max("Welcome", 23); *// compilation error!*
  - *Goal: Make it triggers a compilation error*

Make it safe using bounded type parameter

- <E **extends** Comparable<E>> declares a bounded type parameter

```
public class Max1 {
  public static <E extends Comparable<E>> E max(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

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

- Max.max("Welcome", 23); // compilation error!

Max1.java

Bounded Type Parameter

Task: Write an equalArea method that can compare two geometric objects.
Q: Which implementation will work?

```
public static void main(String[] args) {
 Rectangle rectangle = new Rectangle(2, 2);
 Circle circle = new Circle(2);
 System.out.println("Same area? " + equalArea(rectangle, circle));
}

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

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

BoundedTypeDemo.java

Bounded Type Parameter

Any differences in the usages of these two overloading equalArea methods?

```
public static void main(String[] args) {
 Rectangle rectangle = new Rectangle(2, 2);
 Circle circle = new Circle(2);
 System.out.println("Same area? " + equalArea(rectangle, circle));
}

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

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

We can parameterize the method call to allow only a specific subclass, e.g., *var b = <Rectangle>equalArea(r1,r2);*

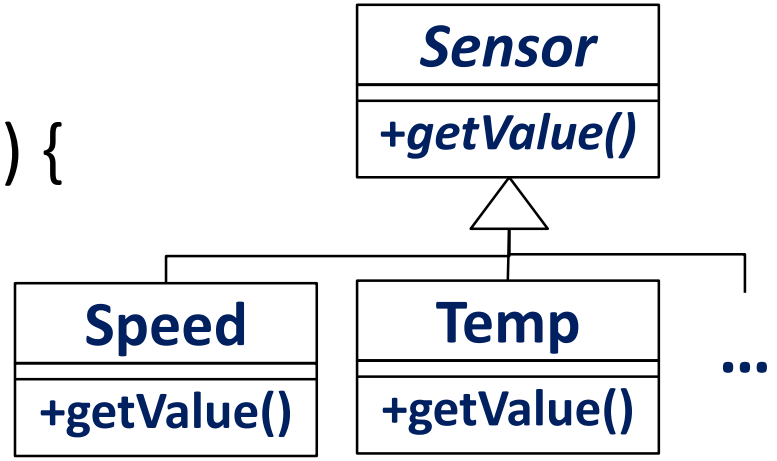
Bounded Type Parameter – A Note for Effective Usage

- Task: There are different types of sensors: speed sensors, temperature sensors, ultrasonic sensors, pressure sensors and so on. Write an equals method to check if two sensors have the same readings

```
static boolean equals (Sensor s1, Sensor s2) {
 return s1.getValue() == s2.getValue();
}
static <E extends Sensor> boolean equals (E s1, E s2) {
 return s1.getValue() == s2.getValue();
}
```

Differences?

SensorDemo.java



Issues in Using a Generic Type

```
public static void main(String[] args ) {
  GenericStack<Integer> intStack = new GenericStack<>();
  intStack.push(1); // 1 is autoboxed into new Integer(1)
  intStack.push(2);
  intStack.push(-2);
  print(intStack);
}
// Print objects and empty stack
public static void print(GenericStack<Object> stack) {
   while (!stack.isEmpty()) {
     System.out.print(stack.pop() + " ");
   }
}
```

Can we use the parameterized type GenericStack<Object> as a polymorphic type?

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list =
  ...
}
```

Issues in Using a Generic Type

```
public static void main(String [] args) {
  var stack = new GenericStack<Integer>();
  func1(stack);
  func2(stack);
  func3(stack);
}
```

Which method call works?

GenericStack is not the same as GenericStack<Object>

GenericStack is equivalent to GenericStack<?>

```
public static void func1(GenericStack<Object> o) { }
public static void func2(GenericStack o) { }
public static void func3(GenericStack<?> o) {  }
```

GenericClassAsType.java

Issues in Using a Generic Type

```
public static void main(String[] args ) {
  GenericStack<Integer> intStack = new GenericStack<>();
  intStack.push(1); // 1 is autoboxed into new Integer(1)
  intStack.push(2);
  intStack.push(-2);
  print(intStack);
}
```

*We can use GenericStack<?> as a **polymorphic type**. Note that GenericStack<E> is not a type on its own.*

```
public static void print(GenericStack<?> stack) {  // GenericStack<?> is a type by itself
  while (!stack.isEmpty()) {
    System.out.print(stack.pop() + " ");
  }
}
```

AnyWildCardDemo.java

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list =
  ...
}
```

Wildcards

- Why wildcards are necessary?
  - Allow using generic classes as **polymorphic types**

	?	unbounded wildcard
➡	? extends T	bounded wildcard
	? super T	lower bounded wildcard

3 forms of using wildcard

  - GenericStack<?> is a polymorphic type because its values such as **new GenericStack<String>()** have more than one type
  - We will discuss the type hierarchy associated with wildcards

```
public static void main(String[] args ) {
  GenericStack<Integer> intStack = new GenericStack<>();
  intStack.push(1); // 1 is autoboxed into new Integer(1)
  intStack.push(2);
  intStack.push(-2);
  System.out.print("The max number is " + max(intStack));
}
// Find the maximum in a stack of numbers
public static double max(GenericStack<Number> stack) {
  double max = stack.pop().doubleValue(); // initialize max
  while (!stack.isEmpty()) {
    double value = stack.pop().doubleValue();
    if (value > max) max = value;
  }
  return max;
}
```

Why bounded wildcards?

Can we use GenericStack<Number> or GenericStack<?> as a type?

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list =
  ...
}
```

WildCardNeedDemo.java

Why bounded wildcards?

```
public static void main(String[] args ) {
  var intStack = new GenericStack<Integer>();
  intStack.push(1); // 1 is autoboxed into new Integer(1)
  intStack.push(2);
  intStack.push(-2);
  System.out.print("The max number is " + max(intStack));
}
// Find the maximum in a stack of numbers
public static double max(GenericStack<? extends Number> stack) {
  double max = stack.pop().doubleValue(); // initialize max
  while (!stack.isEmpty()) {
    double value = stack.pop().doubleValue();
    if (value > max) max = value;
  }
  return max;
}
```

We can use GenericStack<? extends Number> as a type.

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list =
  ...
}
```

WildCardNeedDemo.java

Terminology

- GenericStack<? extends Number> is a bounded generic type
- **? extends Number** is a bounded type parameter
- GenericStack<? extends Number> can be used as a polymorphic type so that its variables or method parameters can reference objects of multiple types in the previous max()

```
var intStack = new GenericStack<Integer>();
var stringStack = new GenericStack<String>();
…
System.out.println(“The max number is “ + max(intStack));
System.out.println(“The largest string is “ + max(stringStack));
```

```
public class SuperWildCardDemo {
public static void main(String[] args) {
  var stack1 = new GenericStack<String>();
  var stack2 = new GenericStack<Object>();
  stack2.push("Java");
  stack2.push(2);
  stack1.push("Sun");
  add(stack1, stack2);
  AnyWildCardDemo.print(stack2);
 }
// add stack1 to stack2
 public static <T> void add(GenericStack<T> stack1, GenericStack<? super T> stack2) {
  while (!stack1.isEmpty())
     stack2.push(stack1.pop());
 }
}
```

Why wildcards with lower bounds?

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list =
  …
}
```

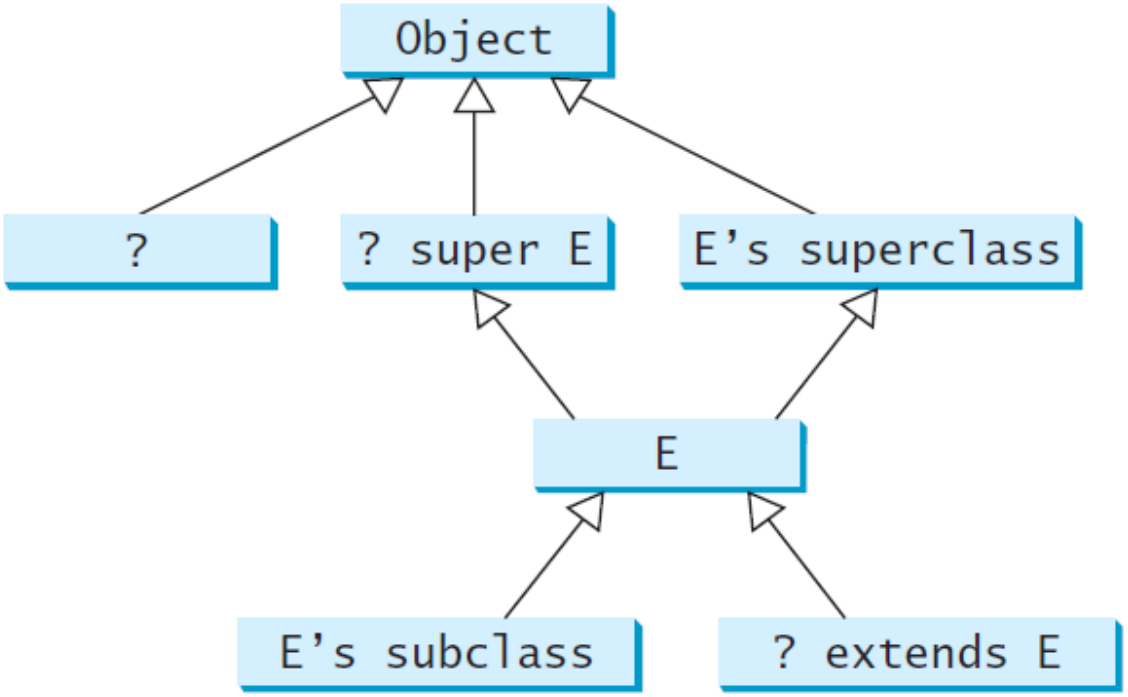
```
public class SuperWildCardDemo {
public static void main(String[] args) {
  var stack1 = new GenericStack<String>();
  var stack2 = new GenericStack<Object>();
  stack2.push("Java");
  stack2.push(2);
  stack1.push("Sun");
  add(stack1, stack2);
  AnyWildCardDemo.print(stack2);
 }
// add stack1 to stack2
 public static <T> void add(GenericStack<? extends T> stack1, GenericStack<T> stack2) {
  while (!stack1.isEmpty())
     stack2.push(stack1.pop());
 }
}
```

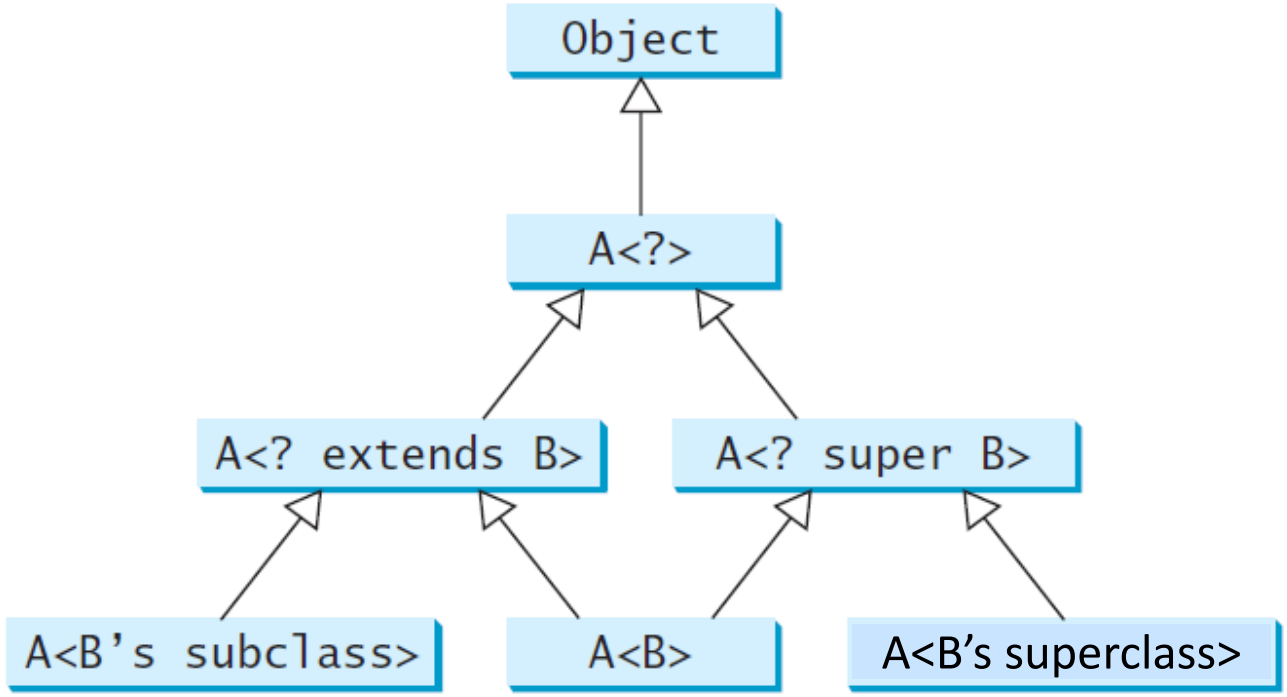
Why wildcards with lower bounds?

```
public class GenericStack<E> {
  private java.util.ArrayList<E> list =
  …
}
```

SuperWildCardDemo.java

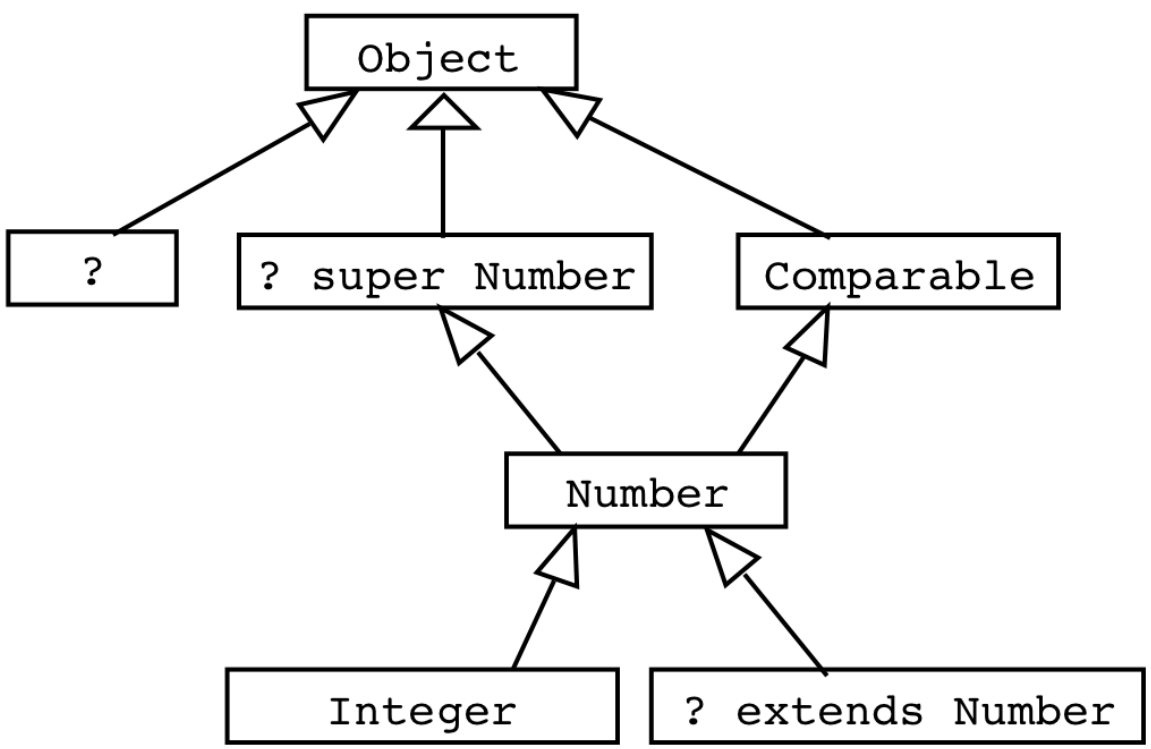
Generic Types and Wildcard Types

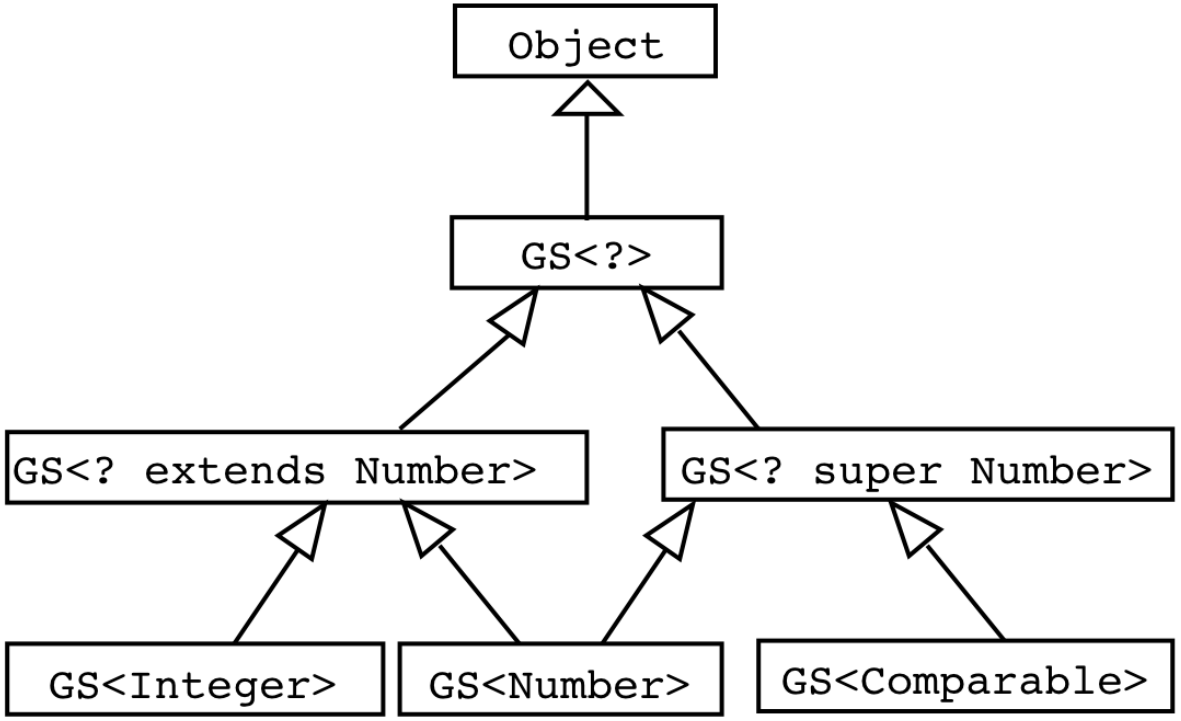




Two type hierarchies defined for generic and wildcard types

Generic Types and Wildcard Types





Wildcard types can be used as values to bind a type parameter, such as E in a generic type GS<E>.

Exercise

```
public class TestPolymorphicType {
    void check() {
        Comparable<?> ca = "welcome";
        Comparable<Object> co = null;
        ✘  co = "hello";
        ✘  co = (Comparable<Object>) "hello";
        ✓  Comparable<? extends String> f = "hello";
        ✓   ca = f;
        ✘   f = ca;
        ✘  co = ca;
        ✓  co = (Comparable<Object>) ca;
        ✓  ca = co;
        ✓  Comparable c = ca;
        ✘  Comparable<Comparable<?>> cc = co;
        ✘  Comparable<Comparable<?>> cc = (Comparable<Comparable<?>>) co;
    }
}
```

TestPolymorphicType.java

Erasure and Restrictions on Generics

- Generics are implemented using an approach called type erasure.
- The generic type information is removed after type checking.
- This approach enables the generics code to be backward-compatible with the legacy code that uses raw types.
  - E.g., Programs using ArrayList were not affected when ArrayList was changed to ArrayList<E> in Java 5.

Compile Time Checking and Type Erasure

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

(a) Original Code

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

(b) Code after Type Erasure

Important Fact

- Note that a generic class is shared by all its instances regardless of its actual generic type.
  - GenericStack<String> stack1 = new GenericStack<>();
  - GenericStack<Integer> stack2 = new GenericStack<>();
- Although GenericStack<String> and GenericStack<Integer> are two types, there is only one class GenericStack loaded into the JVM.

Runtime cannot use generic type information

- What will be printed?

```
public class TestInstanceof<E> {
  public static void main(String [] args) {
    var rawStack = new GenericStack();
    var intStack = new GenericStack<Integer>();
    System.out.println(intStack instanceof GenericStack<Object>);
    System.out.println(intStack instanceof GenericStack);
    System.out.println(intStack instanceof GenericStack<?>);
    System.out.println(rawStack instanceof GenericStack<?>);
  }
}
```

compilation error because the use of generic type is not allowed at runtime

TestInstanceof

Restrictions on Generics 

- Restriction 1: Cannot create an instance of a generic type. (i.e., new E()).
- Restriction 2: Generic array creation is not allowed. (i.e., new E[100]).
- Restriction 3: A generic type parameter of a class is not allowed in a static context.
- Restriction 4: Exception classes cannot be generic.

Not allowed in a static context

```
public class Test<E> {
  public static void m (E o1) { // Illegal
  }
  public static E o1; // Illegal
  static {
    E o2; // Illegal
  }
}
```

There is only one class file for Test,
JVM cannot differentiate at run time:
  Test<String>.o1 // a String object
from
  Test<Date>.o1. // a Date object

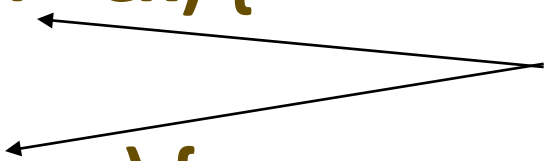
Restrictions on Generics

- Restriction 1: Cannot create an instance of a generic type. (i.e., new E()).
- Restriction 2: Generic array creation is not allowed. (i.e., new E[100]).
- Restriction 3: A generic type parameter of a class is not allowed in a static context.
- Restriction 4: Exception classes cannot be generic.

Not allowed in exception classes

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

try {
  …
} catch (MyException<Integer> ex) {
  …
} catch (MyException<Circle> ex) {
  …
}
```



Generic Constructors

```
public class GenericConstructors {
  ArrayList<? extends Number> list;

  <E extends Number> GenericConstructors(E o) {
    var list = new ArrayList<E>();
    list.add(o);
    this.list = list;
  }

  public static void main(String[] args) {
    System.out.println(new <Integer>GenericConstructors(0).list);
    System.out.println(new <Double>GenericConstructors(0.1).list);
  }
}
```

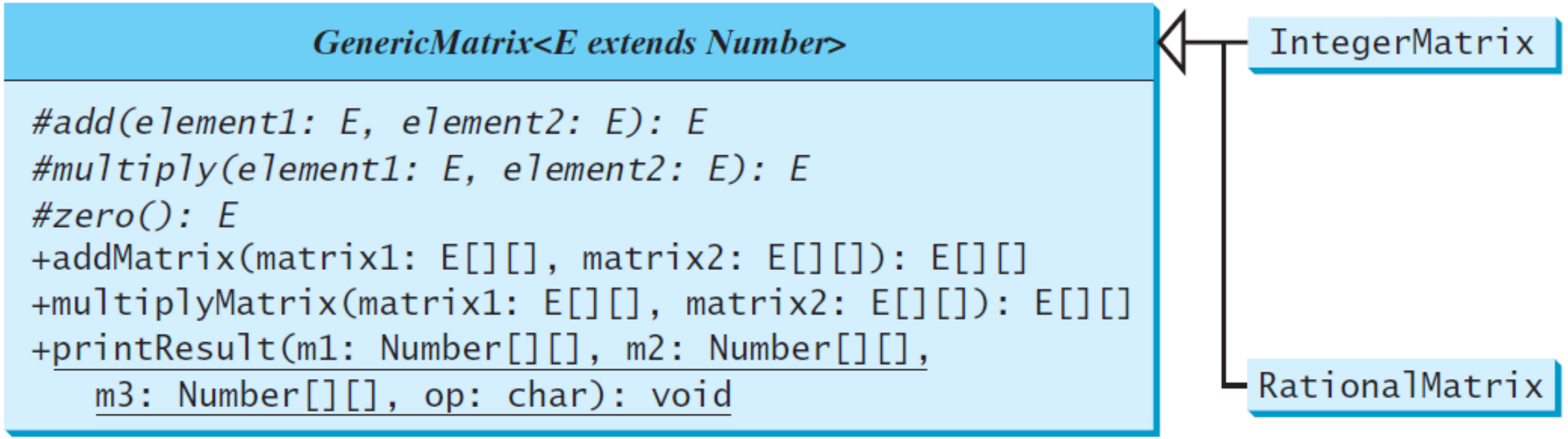
GenericConstructors.java

Practice-1: Designing Generic Matrix Classes

- Task: Write a generic class for matrix arithmetic. The class implements matrix addition and multiplication common for all types of matrices.

GenericMatrix.java

Practice-1: Designing Generic Matrix Classes



Practice-2: Designing Generic Matrix Classes

- Task: Write two programs that utilize the GenericMatrix class for integer matrix arithmetic and rational matrix arithmetic, respectively.

IntegerMatrix.java
TestIntegerMatrix.java
RationalMatrix.java
TestRationalMatrix.java