Java generics

Yulia Newton, Ph.D.
CS151, Object Oriented Programming
San Jose State University
Spring 2020

Introduction to Java generics

- Extension to Java's type system
- Allow a type of the object to be a parameter to a method
  - Allows methods to operate on objects of various types
  - Write a single method and re-use it for many object types
- Similar to templates in C++
- Specified with <>
  - *SuperType <SubType> obj = new SuperType <SubType>();*
  - Type could be a class or an interface
- Generic types cannot be primitives
  - Use the corresponding classes

Motivation for generics

- Code re-use
  - Write a single method for handling multiple object types
  - Write data structures that can store multiple object types
    - E.g. *List* that can store any type of object instead of a List that can store *String* objects, another *List* that can store *Integer* objects, etc.
- Type safety
  - Get compile-time errors for invalid type operations
  - More stable software

Example: motivation for generics with a simple print method

```java
public class Test {
    private static void printObject(String objectToPrint) {
        System.out.print(objectToPrint);
    }

    private static void printObject(Integer objectToPrint) {
        System.out.print(objectToPrint);
    }

    public static void main(String[] args){
        printObject("This is a String");
        System.out.println();

        printObject(42);
        System.out.println();

        printObject(1.2);
        System.out.println();
    }
}
```

We implement two different methods (overloading) for different types of objects to print

Oops! We try to print a Double and there is no method that accepts that type

```
Test.java:3439: error: no suitable method found for printObject(double)
        printObject(1.2);
        ^
    method Test.printObject(String) is not applicable
      (argument mismatch; double cannot be converted to String)
    method Test.printObject(Integer) is not applicable
      (argument mismatch; double cannot be converted to Integer)
1 error
```

Example: motivation for generics with a simple print method

```java
public class Test {
    private static void printObject(String objectToPrint) {
        System.out.print(objectToPrint);
    }

    private static void printObject(Integer objectToPrint) {
        System.out.print(objectToPrint);
    }

    public static void main(String[] args){
        printObject("This is a String");
        System.out.println();

        printObject(42);
        System.out.println();

        printObject(1.2);
        System.out.println();
    }
}
```

```
Test.java:3439: error: no suitable method found for printObject(double)
        printObject(1.2);
        ^
    method Test.printObject(String) is not applicable
      (argument mismatch; double cannot be converted to String)
    method Test.printObject(Integer) is not applicable
      (argument mismatch; double cannot be converted to Integer)
1 error
```

We need to add yet another printObject() method that accepts Double type:

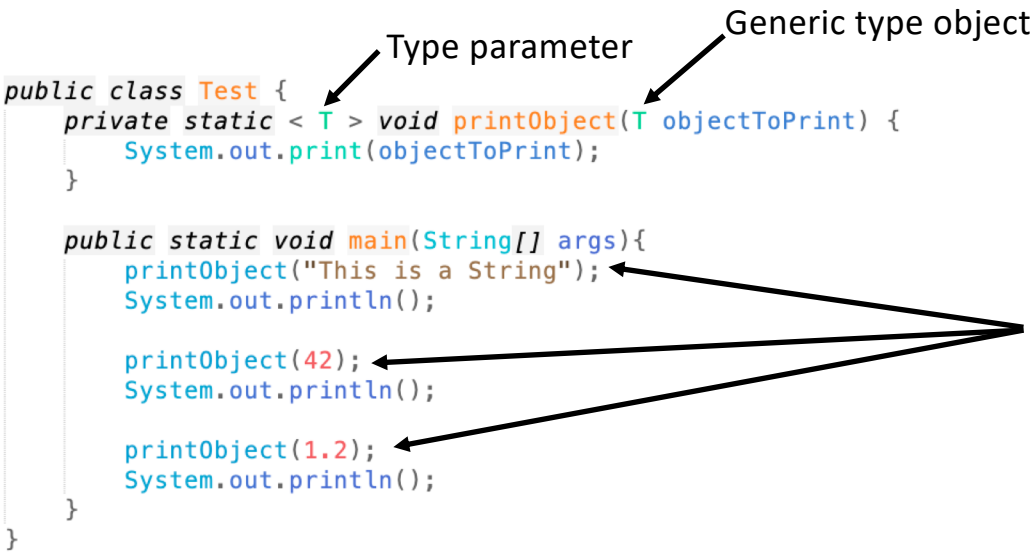
```java
public class Test {
    private static void printObject(String objectToPrint) {
        System.out.print(objectToPrint);
    }

    private static void printObject(Integer objectToPrint) {
        System.out.print(objectToPrint);
    }

    private static void printObject(Double objectToPrint) {
        System.out.print(objectToPrint);
    }

    public static void main(String[] args){
        printObject("This is a String");
        System.out.println();

        printObject(42);
        System.out.println();

        printObject(1.2);
        System.out.println();
    }
}
```

… or we solve this problem by using a generic method

Generic methods

- Type parameter enclosed in <> appears before return type
  - Sometimes called “type variable”
  - Type parameter is a placeholder for actual types passed to the method
  - Type parameter is an identifier for generic type
- Multiple type parameters are separated by a comma
- Method body is the same as if you code a non-generic method

Example: print a generic element



Output:

```
This is a String
42
1.2
```

Objects of different types are passed into printObject() method, no need to write three different versions of this method for each type

Example: printing Employee object

```java
class Employee{
    private String name;
    private int id;
    private Gender gender;

    Employee(){
        this.name = "Uknown name";
        this.id = 0;
        this.gender = Gender.UNSPECIFIED;
    }

    Employee(String name, int id, Gender gender){
        this.name = name;
        this.id = id;
        this.gender = gender;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public Gender getGender(){return this.gender;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setGender(Gender gender){this.gender = gender;}

    @Override
    public String toString(){
        return this.name+" ("+this.id+") is a "+this.gender;
    }
}
```

```java
public class Test {
    private static < T > void printObject(T objectToPrint) {
        System.out.print(objectToPrint);
    }

    public static void main(String[] args){
        Employee e = new Employee("Will Smith", 101, Gender.MALE);
        printObject(e);
        System.out.println();
    }
}
```

Output:

```
Will Smith (101) is a MALE
```

If a class has *toString()* method implemented then an object of that type will display “useful” information

Example: printing Employee object without toString() method overloading

```java
class Employee{
    private String name;
    private int id;
    private Gender gender;

    Employee(){
        this.name = "Uknown name";
        this.id = 0;
        this.gender = Gender.UNSPECIFIED;
    }

    Employee(String name, int id, Gender gender){
        this.name = name;
        this.id = id;
        this.gender = gender;
    }

    public String getName(){return this.name;}
    public int getID(){return this.id;}
    public Gender getGender(){return this.gender;}

    public void setName(String name){this.name = name;}
    public void setID(int id){this.id = id;}
    public void setGender(Gender gender){this.gender = gender;}
}
```

No *toString()* method implementation

```java
public class Test {
    private static < T > void printObject(T objectToPrint) {
        System.out.print(objectToPrint);
    }

    public static void main(String[] args){
        Employee e = new Employee("Will Smith", 101, Gender.MALE);
        printObject(e);
        System.out.println();
    }
}
```

Output:

```
Employee@7852e922
```

toString() method provides informative information about the object; it is recommended to overload this method in your class implementations

Example: testing object equality with generics

Two objects of generic type; no need to implement this method for every object type we might encounter

```
public class Test {
    public static < T > boolean isEqual(T object1, T object2){
        return object1.equals(object2);
    }

    public static void main(String[] args){
        String s1 = "This is a string";
        String s2 = "This is a string";
        System.out.println("s1 == s2: "+isEqual(s1,s2));

        s2 = "This is another string";
        System.out.println("After change s1 == s2: "+isEqual(s1,s2));


        Integer i1 = 10;
        Integer i2 = 10;
        System.out.println("i1 == i2: "+isEqual(i1,i2));

        i2 = 15;
        System.out.println("After change i1 == i2: "+isEqual(i1,i2));
    }
}
```

Output:

```
s1 == s2: true
After change s1 == s2: false
i1 == i2: true
After change i1 == i2: false
```

Example: generic array printing method

Type parameter

Generic type array

```
public class Test {
    private static < E > void displayElements(E[] inputElements) {
        for(E element : inputElements){
            System.out.print(element+" ");
        }
    }

    public static void main(String[] args){
        Integer[] intArray= {1,2,3,4,55,6,7,8,9};
        displayElements(intArray);
        System.out.println();

        Double[] doubleArray = {1.2,2.3,3.4,4.5};
        displayElements(doubleArray);
        System.out.println();

        Character[] charArray = { 'A','B','C','D','E','F','G' };
        displayElements(charArray);
        System.out.println();
    }
}
```

Output:

```
1 2 3 4 55 6 7 8 9
1.2 2.3 3.4 4.5
A B C D E F G
```

Arrays of different types are declared but the same method is called to print them

Java generics parameter type naming convention

- Parameter type naming convention
  - E – Element
  - K - Key
  - N - Number
  - T - Type
  - V - Value
  - S,U,V etc. – additional types
- Guidelines
  - Follow the convention
  - Document in regular and Javadoc comments
  - Can number conventional parameter types
    - <T1, T2, T3>
  - When needed use prefixes or suffixes
    - KeyT, ValueT
  - Many modern IDEs distinguish generic types by color to make it easier to spot them

Example: generic method with multiple type parameters

Separate type parameters by comma

```
public class Test {
    private static < K, V > void printKeyValue(K keyObject, V valueObject) {
        System.out.print(keyObject+": "+valueObject);
    }

    public static void main(String[] args){
        printObject("StringKey",101);
        System.out.println();

        printObject(10,"this is my value");
        System.out.println();

        printObject("StringKey",10.5);
        System.out.println();

        printObject(12,1.2);
        System.out.println();
    }
}
```

Output:

```
StringKey: 101
10: this is my value
StringKey: 10.5
12: 1.2
```

Example: generic method with generic return type

```java
public class Test {
    private static < T > T returnSelf(T myObject) {
        return myObject;
    }

    public static void main(String[] args){
        String myString = "This is a string";
        System.out.println(returnSelf(myString));
        System.out.println();

        Integer myInt = 100;
        System.out.println(returnSelf(myInt));
        System.out.println();

        Double myDouble = 10.5;
        System.out.println(returnSelf(myDouble));
        System.out.println();
    }
}
```

```
This is a string

100

10.5
```

Example: type safety, array of generic types

```java
import java.util.*;

public class Test {
    public static void main(String[] args){
        ArrayList myArray = new ArrayList();

        myArray.add("Dog");
        myArray.add("Cat");
        myArray.add("Frog");
        myArray.add(42);

        System.out.println((String)myArray.get(0));
        System.out.println((String)myArray.get(1));
        System.out.println((String)myArray.get(2));
        System.out.println((String)myArray.get(3));
    }
}
```

Run-time exception:

```
Dog
Cat
Frog
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
        at Test.main(Test.java:3504)
```

```java
import java.util.*;

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

        myArray.add("Dog");
        myArray.add("Cat");
        myArray.add("Frog");
        myArray.add(42);

        System.out.println((String)myArray.get(0));
        System.out.println((String)myArray.get(1));
        System.out.println((String)myArray.get(2));
        System.out.println((String)myArray.get(3));
    }
}
```

Compile-time error:

```
Test.java:3501: error: no suitable method found for add(int)
        myArray.add(42);
                ^
    method Collection.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method List.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method AbstractCollection.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method AbstractList.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method ArrayList.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full output
1 error
```

Generic classes

- Called “parameterized classes”
- Class name is followed by type parameter section
  - *public class Employee <T> {…}*
- Multiple type parameters are allowed
  - Separated by comma
- Generic class declarations look like non-generic class declarations except for generic type usage

Example: generic Rectangle class

Generic type parameter

```
class Rectangle <T> {
    private T width;
    private T length;

    Rectangle(T width, T length){
        this.width = width;
        this.length = length;
    }

    @Override
    public String toString(){
        return this.width+" x "+this.length;
    }
}

public class Test {
    public static void main(String[] args){
        Rectangle r1 = new Rectangle(5,10);
        System.out.println("r1 rectangle: "+r1.toString());

        Rectangle r2 = new Rectangle(2.1,8.9);
        System.out.println("r2 rectangle: "+r2.toString());
    }
}
```

Private attributes of generic type

Output:

```
r1 rectangle: 5 x 10
r2 rectangle: 2.1 x 8.9
```

Instances of the same class type are instantiated with different argument types (Integer and Double)

Example: PriorityQueue that does not allow duplicate elements

```
class PriorityQueueNoDuplicates<E> extends PriorityQueue<E>{
    @Override
    public boolean add(E e)
    {
        boolean isAdded = false;
        if(!super.contains(e)){
            isAdded = super.offer(e);
        }
        return isAdded;
    }
}

public class Test {
    public static void main(String[] args){
        PriorityQueue<String> names = new PriorityQueueNoDuplicates<String>();

        // queue elements:
        names.add("Jeff");
        names.add("Dan");
        names.add("Joe");
        names.add("Jeff");

        Iterator nameIterator = names.iterator();
        while(nameIterator.hasNext()){
            System.out.println(nameIterator.next());
        }
    }
}
```

Generic type of elements

Jeff is not added

Output:

```
Dan
Jeff
Joe
```

Example: class with multiple generic types

Multiple type parameters

```
class Rectangle <T, S> {
    private T width;
    private T length;
    private S label;

    Rectangle(T width, T length, S label){
        this.width = width;
        this.length = length;
        this.label = label;
    }

    @Override
    public String toString(){
        return this.width+" x "+this.length+" ("+this.label+")";
    }
}

public class Test {
    public static void main(String[] args){
        Rectangle r1 = new Rectangle(5,10,"Rectangle1");
        System.out.println("r1 rectangle: "+r1.toString());

        Rectangle r2 = new Rectangle(5,10,101);
        System.out.println("r2 rectangle: "+r2.toString());
    }
}
```

Output:

```
r1 rectangle: 5 x 10 (Rectangle1)
r2 rectangle: 5 x 10 (101)
```

Label field can be instantiated to String or Integer or other types

Instantiating generic types

- The specific type has to be specified when an instance of generic type is being instantiated
  - The type is specified using <> that come after the generic type

Example: instantiating an instance of generic class

Generic class with some getters, setters, and toString() implementation

```
class GenericClass < T > {
    private T myAttribute;

    GenericClass(T t){this.myAttribute = t;}
    public void setAttribute(T t){this.myAttribute = t;}
    public T setAttribute(){return this.myAttribute;}
    public String toString(){return this.myAttribute.toString();}
}


public class Test {
    public static void main(String[] args){
        GenericClass<String> s = new GenericClass("This is my string");
        System.out.println("GenericClass<String>: "+s.toString());

        GenericClass<Integer> i = new GenericClass(10);
        System.out.println("GenericClass<Integer>: "+i.toString());

        GenericClass<Double> d = new GenericClass(2.5);
        System.out.println("GenericClass<Double>: "+d.toString());
    }
}
```

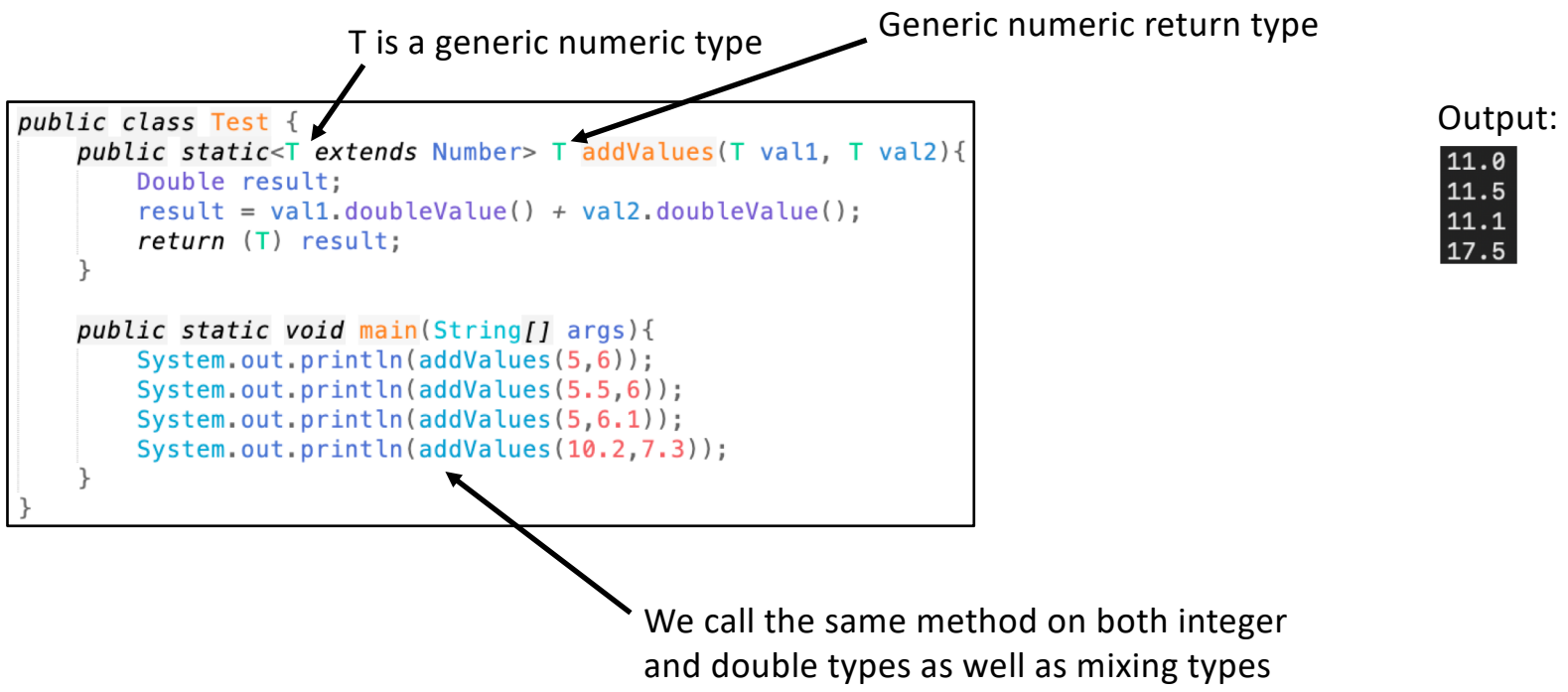
Output:

```
GenericClass<String>: This is my string
GenericClass<Integer>: 10
GenericClass<Double>: 2.5
```

Bounded vs. non-bounded generics

- There are times when we want to restrict the types of object that can be used in parameterized type
  - E.g. a method can only take numeric types
- Bind type parameter by adding "*extends*" keyword followed by the type representing the upper bound
  - <T extends Number>
  - Keyword *extends* is used for both classes and interfaces when specifying bounds
- Binding causes compile-time errors when an invalid type is used
- Multiple bounds can be used
  - <T extends A & B & C>
  - Not allowed to have multiple classes specified in multiple bounds, but allowed to use multiple interfaces
    - E.g. A is a class, but B and C are interfaces
    - E.g. B is a class, but A and C are interfaces
    - E.g. C is a class, but A and B are interfaces

Example: generic method to add two numeric values



Output:

```
11.0
11.5
11.1
17.5
```

Example: compare objects

Type T must be a subtype of Comparable interface

Method to compare two objects

```
public class Test {
    public static < T > boolean isEqual(T object1, T object2){
        return object1.equals(object2);
    }

    public static <T extends Comparable> int compare(T object1, T object2){
        return object1.compareTo(object2);
    }

    public static void main(String[] args){
        String s1 = "This is a string";
        String s2 = "This is a string";
        System.out.println("s1 compare to s2: "+compare(s1,s2));

        s2 = "This is another string";
        System.out.println("After change s1 compare to s2: "+compare(s1,s2));


        Integer i1 = 10;
        Integer i2 = 10;
        System.out.println("i1 compare to i2: "+compare(i1,i2));

        i2 = 15;
        System.out.println("After change i1 compare to i2: "+compare(i1,i2));
    }
}
```

Output:

```
s1 compare to s2: 0
After change s1 compare to s2: -78
i1 compare to i2: 0
After change i1 compare to i2: -1
```

Similar to previous *isEqual()* example

Example: print array of Animal type

```
abstract class Animal{}

class Dog extends Animal{
    public String toString(){return "I am a dog";}
}

class Cat extends Animal{
    public String toString(){return "I am a cat";}
}

class Frog extends Animal{
    public String toString(){return "I am a frog";}
}

public class Test {
    private static < E extends Animal> void displayElements(E[] inputElements) {
        for(E element : inputElements){
            System.out.print(element.toString()+"\n");
        }
    }

    public static void main(String[] args){
        Dog d1 = new Dog();
        Dog d2 = new Dog();
        Cat c1 = new Cat();
        Frog f1 = new Frog();

        Animal[] myArray= {d1,d2,c1,f1};
        displayElements(myArray);
    }
}
```

Output:

```
I am a dog
I am a dog
I am a cat
I am a frog
```

Example: type safety with the previous example

```
abstract class Animal{}

class Dog extends Animal{
    public String toString(){return "I am a dog";}
}

class Cat extends Animal{
    public String toString(){return "I am a cat";}
}

class Frog extends Animal{
    public String toString(){return "I am a frog";}
}

class Shape{
    public String toString(){return "I am a shape";}
}

public class Test {
    private static < E extends Animal> void displayElements(E[] inputElements) {
        for(E element : inputElements){
            System.out.print(element.toString()+"\n");
        }
    }

    public static void main(String[] args){
        Dog d1 = new Dog();
        Dog d2 = new Dog();
        Cat c1 = new Cat();
        Frog f1 = new Frog();
        Shape s = new Shape();

        Animal[] myArray= {d1,d2,c1,f1,s};
        displayElements(myArray);
    }
}
```

Compile time error:

```
Test.java:3671: error: incompatible types: Shape cannot be converted to Animal
        Animal[] myArray= {d1,d2,c1,f1,s};
                                       ^
1 error
```

We add new Shape class that is NOT a child of Animal

New instance of Shape

Add instance of Shape to myArray

Example: max method for array of comparable types

```java
public class Test {
    private static < E extends Comparable> E max(E[] inputElements) {
        E currentMax = inputElements[0];
        for(E element : inputElements){
            if(element.compareTo(currentMax) > 0){
                currentMax = element;
            }
        }
        return currentMax;
    }

    public static void main(String[] args){
        Integer[] intArray= {1,9,3,4,8,0};
        System.out.println(max(intArray));

        Double[] doubleArray= {11.1,9.2,3.4,3.9,7.9,0.3};
        System.out.println(max(doubleArray));

        String[] stringArray= {"Apple","Plum","Watermellon","Pineapple"};
        System.out.println(max(stringArray));
    }
}
```

Return the max value of the input array

Output:

```
9
11.1
Watermellon
```

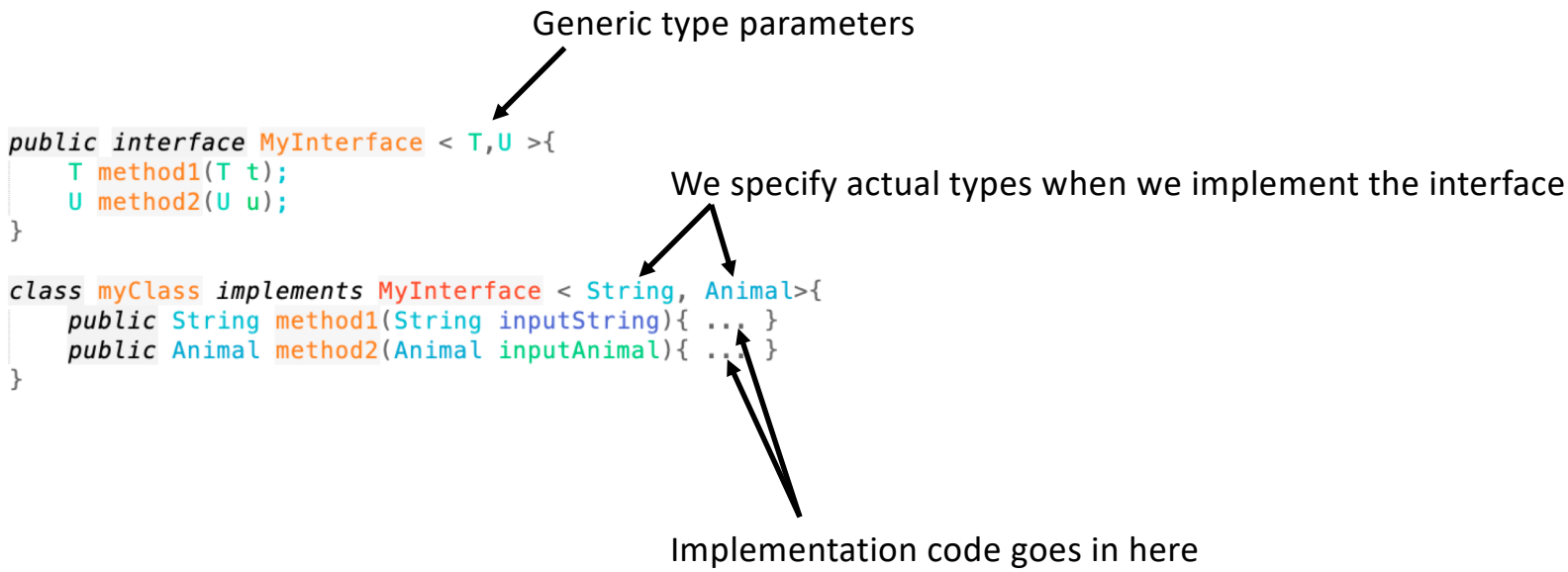
Obtain max elements of *Integer*, *Double*, and *String* arrays

Generic interfaces

- We can use generic types with interfaces
  - E.g. *Comparable* built-in Java interface

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

Example: using generics with interfaces



Subtyping with Java generics

- *Object* is a superclass of every type in Java
- *List<Object>* is not a superclass of *List<String>*

This code works:

```
public class Test {
    public static void main(String[] args){
        Object myObject = new Object();
        Integer myInt = 5;
        myObject = myInt;

        List<String> lst1 = new ArrayList<String>();
        List<Object> lst2 = new ArrayList<Object>();
        myObject = lst1;
        //lst2 = lst1;
    }
}
```

Valid

Compile error:

```
public class Test {
    public static void main(String[] args){
        Object myObject = new Object();
        Integer myInt = 5;
        myObject = myInt;

        List<String> lst1 = new ArrayList<String>();
        List<Object> lst2 = new ArrayList<Object>();
        myObject = lst1;
        lst2 = lst1;
    }
}
```

Invalid

```
Test.java:3733: error: incompatible types: List<String> cannot be converted to List<Object>
        lst2 = lst1;
               ^
1 error
```

Subtyping with Java generics (cont'd)

- If we have a generic type T and B is a subtype of A it is not true that T<B> is a subtype of T<A>
  - *class B extends A {…}*
  - *T<A> a = new T<A>();*
  - *T<B> b = new T<B>();*
  - *a = b; // compile error*

Wildcards in Java generics

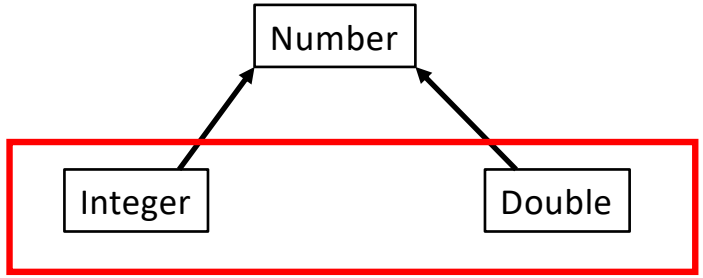
- Wildcards in Java are used when type is unknown
  - Represented by a question mark
- Wildcards can be used in method parameters, class attributes, local variables, and return types
- Convenient to use in collections
- Wildcards are indicated with ? character (question mark character)
  - E.g. *ArrayList<?>*

Types of wildcards

- Upper bounded wildcards
  - Allows relaxing restrictions on the variable type
  - Use key word *extends*
  - E.g. if we want to restrict variables to numeric types we can use *List<? extends Number>*
    - Works for *integer*, *double*, *number*
- Lower bounded wildcards
  - Use keyword *super*
  - E.g. if we want to restrict variables to integers and integer parent (Number)
- Unbounded wildcards
  - Simply use ?
    - Unknown type

Example: using upper bounded wildcard

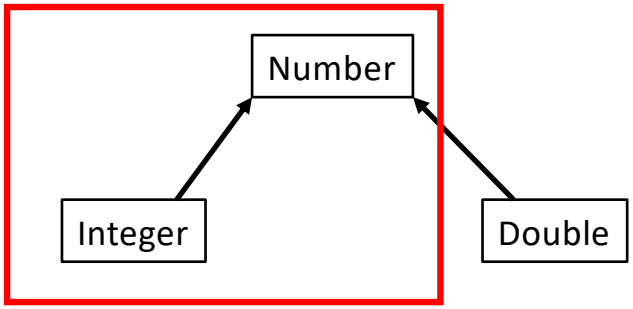
```java
//Java program to demonstrate Upper Bounded Wildcards
import java.util.Arrays;
import java.util.List;

class WildcardDemo
{
    public static void main(String[] args)
    {

        //Upper Bounded Integer List
        List<Integer> list1= Arrays.asList(4,5,6,7);

        //printing the sum of elements in list
        System.out.println("Total sum is:"+sum(list1));

        //Double list
        List<Double> list2=Arrays.asList(4.1,5.1,6.1);

        //printing the sum of elements in list
        System.out.print("Total sum is:"+sum(list2));
    }

    private static double sum(List<? extends Number> list)
    {
        double sum=0.0;
        for (Number i: list)
        {
            sum+=i.doubleValue();
        }

        return sum;
    }
}
```



www.geeksforgeeks.org

Example: using lower bounded wildcard

```java
//Java program to demonstrate Lower Bounded Wildcards
import java.util.Arrays;
import java.util.List;

class WildcardDemo
{
    public static void main(String[] args)
    {
        //Lower Bounded Integer List
        List<Integer> list1= Arrays.asList(4,5,6,7);

        //Integer list object is being passed
        printOnlyIntegerClassorSuperClass(list1);

        //Number list
        List<Number> list2= Arrays.asList(4,5,6,7);

        //Integer list object is being passed
        printOnlyIntegerClassorSuperClass(list2);
    }

    public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list)
    {
        System.out.println(list);
    }
}
```



www.geeksforgeeks.org

Example: using unbounded wildcard

```
//Java program to demonstrate Unbounded wildcard
import java.util.Arrays;
import java.util.List;

class unboundedwildcardemo
{
    public static void main(String[] args)
    {

        //Integer List
        List<Integer> list1= Arrays.asList(1,2,3);

        //Double list
        List<Double> list2=Arrays.asList(1.1,2.2,3.3);

        printlist(list1);

        printlist(list2);
    }

    private static void printlist(List<?> list)
    {

        System.out.println(list);
    }
}
```

www.geeksforgeeks.org

Type erasure

- Compiler uses “type erasure” procedure at compile time
- Type erasure
  - All generic types are implemented as type *Object* in unbounded generic type
  - Generic types are implemented as the specified upper bound if bounded generic type
- Why we cannot use primitives in Java generic types?
  - Primitive types do not extend *Object* type
  - Use corresponding classes that are subtypes of *Object* instead

Concluding remarks

- Java generics is a powerful tool that makes coding easier and less error-prone

- Advantages of Java generics
  - Code re-use for classes, interfaces, and methods
  - Type errors are caught at compile time
    - Compile time type safety