Mahidol University
Faculty of Information
and Communication Technology

LECTURE 09
Generic Type

ITCS123 ObjectOriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

Semester 2/2023 14 Mar 2024

Learning Objectives

- To understand the objective of **generic programming**
- To be able to implement **generic classes** and **generic methods**
- To know how to use **bounded type parameters**
- To understand the relationship between generic types and inheritance

Source: https://slideplayer.com/slide/12771077/

“Generics” = “Parameterized Types”

- Generics let you create classes, interfaces, and methods that work in the same way on **different types** of objects
- The term generic comes from the idea that we would like to be able to write **general algorithms*** that can be broadly reused for many types of objects rather than having to adapt our code to fit each circumstance
- Generics take **reuse** to the next level by making the **type of the objects** we work with an **explicit parameter** of the generic code.
  → **Parameterized Type**
- A class, interface, or method that operates on a parameterized type is called ***generic***, as in ***generic class*** or ***generic method***

**More about algorithms in your future course during your second year =D*

Same Code for Many Types of Objects

ArrayList <E>

```
-----------------------------
void add(int index, E elemen)
boolean add(E element)
E get(int index)
E remove(int index)
E set(int index, E element)
int size()
...
```

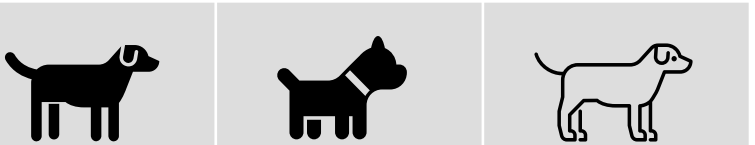
ArrayList <Integer>

5	10	15

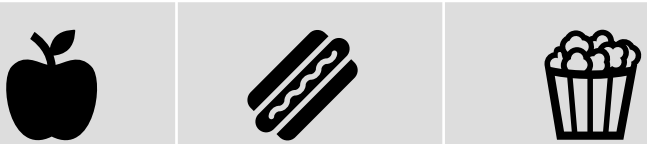
ArrayList <String>

Alice	Bob	Cindy

ArrayList <Dog>



ArrayList <Item>



Generic Classes

- A class with one or more type parameters. A class that can work with any data types
- We have used that already. Something with **angle brackets** < …. >

```
ArrayList<String> names = new ArrayList<String>();
```

- The type **ArrayList<String>** denotes an array list of String elements. The angle brackets around the String type tell you that String is a **type parameter**. You can replace String with any other class and get a different array list type.

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

- For that reason, **ArrayList** is called a **generic class**.

Example Usage

```
ArrayList<String> names = new ArrayList<String>();
names.add("java");
Cat myCat = new Cat("Kitty");     // suppose kitty is a cat's name
names.add(myCat);                 // error: Cat is not a String type
names.add(myCat.name);            // OK: Cat's name is a String type
```

Sorry Kitty,
You are not String.
You are not fit here!

```
ArrayList<Object> objects = new ArrayList<Object>();     // list of anything
objects.add("java");
objects.add(myCat);               // OK: Cat is an Object
                                  // (every class extends Object class)
```

Why Generic Programming?

- Since Java 5, generics have been a part of the language.
- Before generics, you had to cast every object you read from a collection
  - uncertain and not typesafe
  - Errors will be found only at *runtime*
  - Violate FAIL FAST principle

```
// Before generic classes
ArrayList names = new ArrayList();

// Since you do not know which kind of object is stored in the list,
// so you have to cast it first
String name = (String) names.get(0);
```

Why Generic Programming? (Cont)

- **Generics Solution: type parameters**
  - The compiler can perform typesafe checks
    - If the wrong type is inserted into the list, you will get the error at *compile time*

```
// After generic classes
ArrayList<String> names = new ArrayList<String>();

// Since this list can contain only a String, you do not need to cast the object
String name = (String) names.get(0);
```

  - Generics make the code ***safer*** and ***easier to read***

- **Generic Programming** means writing code that can be reused for objects of many different types

Create your own Generic Class

```
public class Box {  //box of an object
    private Object object;
    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

One box fits ALL

```
public class Box { //box of a string
    private String object;
    public void set(String object) { this.object = object; }
    public String get() { return object; }
}
```

Specific type of box

```
public class Box<T> { // box of something
                      // (have to specify type when declare & construct)
    // T stands for "Type"
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Support various sizes of box. Once identify, only that type can fit it

Create your own Generic Class (Cont)

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a **type parameter section**.
- The type parameter section of a generic class can have one or more type parameters separated by commas

Note that type parameters can represent only **reference types**, not primitive types (like int, double, and char)

Type parameter section

```
public class Box <T> {  // box of something

    private T t;     // T stands for "Type"
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

An Object of Generic Class

- You have to always provide **type parameter**

```
public class Box<T> { // box of something (have to specify when declare and construct)

    private T t;          // T stands for "Type"
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

```
Box box = new Box();  // ERROR
```

```
Box is a raw type. References to generic type Box<T>
should be parameterized
```

```
Box<String> stringBox = new Box<String>();      // OK: Parameter type is String

stringBox.set("Alice");
String item = stringBox.get();
System.out.println(item); // Alice
```

```
Box<Integer> intBox = new Box<Integer>(); // OK
intBox.set(10);
int x = intBox.get();
System.out.println(x); //10
```

Generic Class with Two Type Parameters

- You can declare more than one type parameter in a generic type
- To Specific two or more type parameters, simply use a comma-separated list
  - For example, `public class KeyValue <K, V>`

Type parameter section

```
public class KeyValue <K, V> {
    private K key;
    private V value;

    public KeyValue(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    public void setKey(K key) {
        this.key = key;
    }
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
}
```

```
public static void main(String[] args) {
    KeyValue<String, Integer> ageMap =
      new KeyValue<String, Integer>("Alice", 19);
    KeyValue<String, String> phoneMap =
      new KeyValue<String, String>("Bob", "123-456");
    System.out.println("Name: " + ageMap.getKey()
      + ", Age: " + ageMap.getValue());
    System.out.println("Name: " + phoneMap.getKey()
      + ", Phone: " + phoneMap.getValue());
}
```

OUTPUT

```
Name: Alice, Age: 19
Name: Bob, Phone: 123-456
```

In Java 7 and up, you don't have to repeat type parameters in the constructor. E.g.,

```
KeyValue<String, Integer> ageMap = new KeyValue<>("Alice", 19);
```

Let's explain

Integer (reference type), NOT int (primitive type)

```
KeyValue<String, Integer> ageMap = new KeyValue<String, Integer>("Alice", 19);
KeyValue<String, String> phoneMap = new KeyValue<String, String>("Bob", "123-456");
```

```
public class KeyValue <K, V> {
    private K key;
    private V value;

    public KeyValue(K key, V value) {
        this.key = key;
        this.value = value;
    }

    . . .
}
```

For the **ageMap** object,

- String is substituted for K, and Integer is substituted for V
- During the construction, the string "Alice" is assigned to the variable `this.key`, and the number 19 is assigned to the variable `this.value`.

For the **phoneMap** object,

- String is substituted for K, and String is also substituted for V
  - Although the type parameters differ in this example (K and V), the type of both arguments can be the same (String and Sring).
- During the construction, the string "Bob" is assigned to the variable `this.key`, and the string "123-456" is assigned to the variable `this.value`.

Let's explain

```
System.out.println("Name: " + ageMap.getKey() + ", Age: " + ageMap.getValue());
System.out.println("Name: " + phoneMap.getKey() + ", Phone: " + phoneMap.getValue());
```

```
public class KeyValue <K, V> {
    private K key;
    private V value;
    . . .

    public K getKey() {
        return key;
    }
    public void setKey(K key) {
        this.key = key;
    }
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
}
```

For the **ageMap** object, K = String, and V = Integer

- ageMap.getKey() will return the string "Alice"
- ageMap.getValue() will return the integer value 19

For the **phoneMap** object, K = String, and V = String

- phoneMap.getKey() will return the string "Bob"
- phoneMap.getValue() will return the string "123-456"

As you can see, the same method may return different data types. This depends on the type parameters you declare for each object during the declaration and construction.

Type Parameter Naming Conventions

- Type parameter names are single, UPPERCASE letters
  - This stands in sharp contrast to the variable naming conventions, so it would be easier to distinguish between a type variable and an ordinary class name.
- The most commonly used type parameter names are
  - E – Element (used extensively by the Java Collections framework)
  - K – Key
  - V – Value
  - N – Number
  - T – Type

Generic Methods

- Single methods can be defined as generics
  - Type parameter section delimited by angle brackets (< and >) that precedes the method's return type
- Similar to a generic class, each **type parameter section** contains one or more type parameters separated by commas.
- The type parameters can be used to declare the **return type** and act as placeholders for the **types of arguments** passed to the generic method which are known as actual **type arguments**
- A generic method's body is declared like that of any other method.
- These methods can be placed inside both generic and ordinary classes.

```
public static void main(String[] args) {
    Integer[] intArray = {1,2,3,4,5};
    Double[] doubleArray = {1.2, 3.5, 4.2, 9.8, 9.9};
    Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
    System.out.println("Print integer array");
    printArray(intArray);
    System.out.println("\nPrint double array");
    printArray(doubleArray);
    System.out.println("\nPrint character array");
    printArray(charArray);
}
```

OUTPUT

```
Print integer array
1 2 3 4 5
Print double array
1.2 3.5 4.2 9.8 9.9
Print character array
H E L L O
```

- Suppose we would like to call **printArray** method to print each element in the array one by one with a space in between
- If we have a variety of array types, then we left with two option
- **Option 1:** we may need to create several overloading methods for each type of input array
- **Option 2:** using a generic method – with a type parameter

```
// without a generic method, we have to overload methods
public static void printArray(Integer[] arr) {
    for(int a : arr) {
        System.out.printf("%s ", a);
    }
}
public static void printArray(Double[] arr) {
    for(double a : arr) {
        System.out.printf("%s ", a);
    }
}
public static void printArray(Character[] arr) {
    for(char a : arr) {
        System.out.printf("%s ", a);
    }
}
```

Option 1

Type parameter section

Type argument

```
// with a generic method, you will need to create only one method
public static <E> void printArray(E[] arr) {
    for(E a : arr) {
        System.out.printf("%s ", a);
    }
}
```

Option 2

Type parameter section

Return type

Type argument

```
public static <E> E getMiddle(E[] list) {
    int mid = list.length / 2;
    return list[mid];
}
public static void main(String[] args) {
    Integer[] intArray = {1,2,3,4,5};
    Double[] doubleArray = {1.2, 3.5, 4.2, 9.8, 9.9};
    Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
    System.out.println("Middle of integer array");
    System.out.print(getMiddle(intArray));
    System.out.println("\nMiddle of double array");
    System.out.print(getMiddle(doubleArray));
    System.out.println("\nMiddle character array");
    System.out.print(getMiddle(charArray));
}
```

Another Example

OUTPUT

```
Middle of integer array
3
Middle of double array
4.2
Middle character array
L
```

Type Parameters / Type Variables Bounds

- Let's think about another algorithm, to find the element of any type with the smallest value in the list.
- Is there a limitation on the type of input parameter, we can take?
  - Can we take a list of integers and find the element with the smallest value?
  - How about double? Character? String? Person? Cat? Dog? Student? etc.
- To be able to find the smallest element, first we have to make sure that our elements are **comparable** to each other.
- This is the role of **Type Parameters Bounds**

Type Parameters Bounds

- To ensure that only classes that implement the Comparable interface are plugged in for T, define a boundary as follows

```
// for a generic method
public static <E extends Comparable<E>> E getSmallest(E[] list)

// for a generic class
public class Box <T extends Comparable<E>>
```

- **extends** `Comparable` serves as a bound on the type parameter `E`.
- Any attempt to plug in a type for `E` that does not implement the `Comparable` interface will result in a compiler error message.
- A bound on a type parameter may be both **class** or **interface**

```
public static <E extends Comparable<E>> E getSmallest(E[] list) {
    if(list == null || list.length == 0)
        return null;
    else {
        E smallest = list[0];
        for(int i = 1; i < list.length; i++) {
            if(smallest.compareTo(list[i]) > 0)
                smallest = list[i];
        }
        return smallest;
    }
}
public static void main(String[] args) {
    Integer[] intArray = {1,2,3,4,5};
    Double[] doubleArray = {1.2, 3.5, 4.2, 9.8, 9.9};
    Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
    System.out.println("Smallest element of integer array");
    System.out.print(getSmallest(intArray));
    System.out.println("\nSmallest element of double array");
    System.out.print(getSmallest(doubleArray));
    System.out.println("\nSmallest element character array");
    System.out.print(getSmallest(charArray));
}
```

OUTPUT

```
Smallest element of integer array
1
Smallest element of double array
1.2
Smallest element character array
E
```

```
class Student{
    private int id;
    private String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
    public String toString() {
        return "id: " + this.id + ", name: " + this.name;
    }
}
```

```
public static void main(String[] args) {
    Student[] studentArray = {
        new Student(888, "Alice"),
        new Student(123, "Bob"),
        new Student(222, "Cindy")
    };
    System.out.println("\nSmallest element student array");
    System.out.print(getSmallest(studentArray));    // ERROR!!!
}
```

Since `Student` class **does not** implement `Comparable` interface, the program has no idea how to compare two or more students to find the student with the smallest value.

Comparable<T> Interface

- The interface Comparable<T> contains a method that can be used to compare one object to another:
  - Note. As you can see <T>, the Comparable is also a generic class

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

- Comparable interface has a **compareTo** method -> `x.compareTo(y)`
  - Return a negative number if x should come before y (or x is less than y),
  - Return zero (0) if x and y are the same,
  - Return a positive number if x should come after y (or x is larger than y)

Examples of Comparable

`Comparable<Integer>`

```
Integer int0 = 0;
Integer int1 = 1;
assert int0.compareTo(int1) < 0;  // since 0 precedes 1
```

`Comparable<String>`

```
String str0 = "zero";
String str1 = "one";
assert str0.compareTo(str1) > 0; // since "zero" follows "one"
```

The type parameter to the interface allows nonsensical comparisons to be caught at compile time:

```
Integer i = 0;
String s = "one";
assert i.compareTo(s) < 0;     // compile-time error
```

Implement Comparable Interface

- The Comparable<T> interface gives fine control over what can and cannot be compared.
- Say that we have a `Fruit` class with subclasses `Apple` and `Orange`. Depending on how we set things up, we may prohibit the comparison of apples with oranges or we may permit such comparison.

```
class Fruit {...}
class Apple extends Fruit implements Comparable<Apple> {...}
class Orange extends Fruit implements Comparable<Orange> {...}
```

- Since `Apple` implements `Comparable<Apple>`, it is clear that you can compare apples with apples, but not with oranges.

Implement Comparable Interface - Student

- Let's try to compare student based on their student ID

```
class Student implements Comparable<Student>{
    private int id;
    private String name;
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
    @Override
    public int compareTo(Student s) {
        if(this.id < s.id)
            return -1; // this ID is precedes s.id
        else if(this.id == s.id)
            return 0; // this ID is the same as s.id
        else
            return 1; // this ID is follows s.id
    }
}
```

```
// Example in main
Student x = new Student(456, "Alice");
Student y = new Student(123, "Bob");
System.out.println(x.compareTo(y)); // 1
```

Now, `Student` is `Comparable`, we can use the Generic Method `getSmallest` with bounded type parameter

```
class Student implements Comparable<Student>{
     private int id;
     private String name;

     public Student(int id, String name) {
        this.id = id;
        this.name = name;
     }
     public String toString() {
        return "id: " + this.id + ", name: " + this.name;
     }

     @Override
     public int compareTo(Student s) {
          if(this.id < s.id)
               return -1; // this ID is precedes s.id
          else if(this.id == s.id)
               return 0; // this ID is the same as s.id
          else
               return 1; // this ID is follows s.id
     }
}
```

```
public static <E extends Comparable<E>> E getSmallest(E[] list) {
     if(list == null || list.length == 0)
          return null;
     else {
          E smallest = list[0];
          for(int i = 1; i < list.length; i++) {
               if(smallest.compareTo(list[i]) > 0)
                    smallest = list[i];
          }
          return smallest;
     }
}
```

```
public static void main(String[] args) {
     Student[] studentArray = {
          new Student(888, "Alice"),
          new Student(123, "Bob"),
          new Student(222, "Cindy")
     };
     System.out.println("Smallest element student array");
     System.out.print(getSmallest(studentArray));
}
```

OUTPUT

```
Smallest element student array
id: 123, name: Bob
```

Type Parameters Bounds: Multiple Bounds

- It is possible to use multiple bounds, but at most one bound can be a class
- For example, from the **`getSmallest`** generic method, if we would like to allow objects that are subclasses of **`Number`** class *and* are **`Comparable`**

```
public static <E extends Number & Comparable<E>> E getSmallest(E[] list) {

    if(list == null || list.length == 0)
        return null;
    else {
        E smallest = list[0];
        for(int i = 1; i < list.length; i++) {
            if(smallest.compareTo(list[i]) > 0)
            smallest = list[i];
        }
        return smallest;
    }
}
```

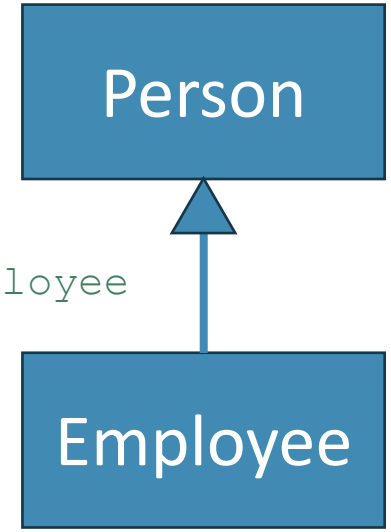
- If the type we are providing extends only one of these two bounds, this will throw an error

```
public static void main(String[] args) {
    Integer[] intArray = {1,2,3,4,5};
    Double[] doubleArray = {1.2, 3.5, 4.2, 9.8, 9.9};
    Character[] charArray = {'H', 'E', 'L', 'L', 'O'};
    Student[] studentArray = {
        new Student(888, "Alice"),
        new Student(123, "Bob"),
        new Student(222, "Cindy")
    };
    System.out.println("Smallest element of integer array");
    System.out.print(getSmallest(intArray));          // OK

    System.out.println("\nSmallest element of double array");
    System.out.print(getSmallest(doubleArray));       // OK

    System.out.println("\nSmallest element character array");
    System.out.print(getSmallest(charArray));         // Compile-Error:
                                                      // Character is comparable but it is not a NUMBER

    System.out.println("\nSmallest element student array");
    System.out.print(getSmallest(studentArray));      // Compile-Error:
                                                      // Student is comparable but it is not a NUMBER
}
```

Inheritance with Generic Classes

- A generic class can be defined as a derived class of an ordinary class or of another generic class
- Generic classes can extend or implement other generic types
  - For example, `ArrayList<T> implements List<T>`
    - So, an `ArrayList<Person>` is subtype of `List<Person>`
- But, given two classes: **A** and **B**, and given **G**: a generic class, there is no relationship between **G<A>** and **G<B>**
  - This is true regardless of the relationship between class A and B
    - E.g., if class **B** is a subclass of class **A**

```
public class Node <T> {
    private T data;
    public void setNode(T data) {
        this.data = data;
    }
    public T getNode() {
        return this.data;
    }
    public static void main(String[] args) {
        Person a = new Person("Alice");
        Employee b = new Employee(123, "Bob");
        Person c = b; // This is okay, since Person is a superclass of Employee

        Node<Person> personNode = new Node<Person>();
        personNode.setNode(new Person("Alice"));
        Node<Employee> empNode = new Node<Employee>();
        empNode.setNode(new Employee(123, "Bob"));
        Node<Person> superNode = empNode; // Illegal
    }
}
```



There is no relationship between
Node<Person> and Node<Employee>,
no matter how Person and Employee are related

Type Parameters Restriction and Limitations

- Type parameters cannot be instantiated with the primitive types (int, double, char, etc.)
- A constructor in a generic class has no type parameter in < >

```
public Box<T>(){ . . . }   // WRONG
public Box(T x){ . . . }   // Correct
```

- However, when a generic class is instantiated, the <> is used.

```
Box<String> box = new Box <String>("A1"); // CORRECT
```

- Type parameters cannot be used as a constructor name or like a constructor

```
T Object = new T(); // Error
T[] a = new T[10];  // Error
```

- Array of a generic class is illegal, but ArrayList is Okay

```
Box<String>[] boxes = new Box<String>[2];   // Error
ArrayList<Box<String>> boxList = new ArrayList<Box<String>>(); // OK
```

Tip: Diamond Syntax

- *Java 7* introduces a convenient syntax enhancement for declaring array lists and other generic classes.
- In a statement that declares and constructs an array list, you need not repeat the type parameter in the constructor. That is, you can write

```
ArrayList<String> names = new ArrayList<>();
```

instead of

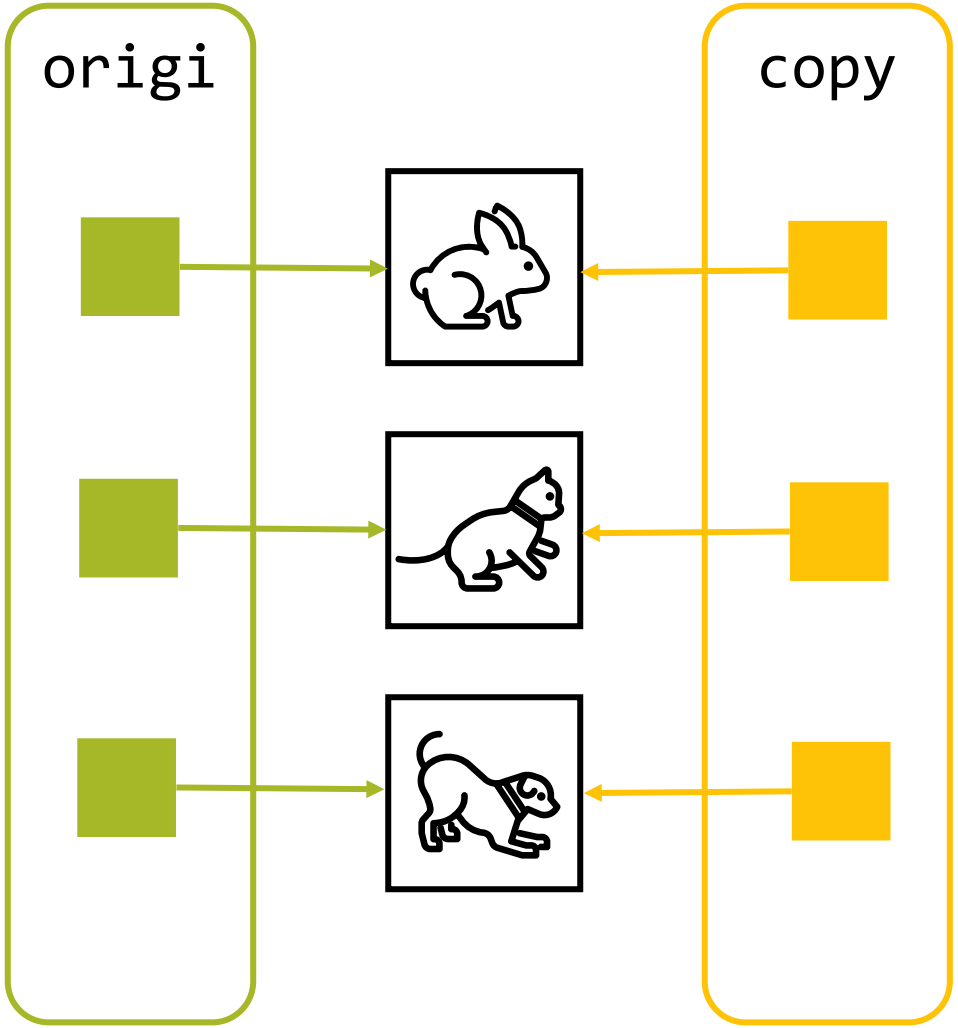
```
ArrayList<String> names = new ArrayList<String>();
```

- This shortcut is called the "diamond syntax" because the empty brackets <> look like a diamond shape.

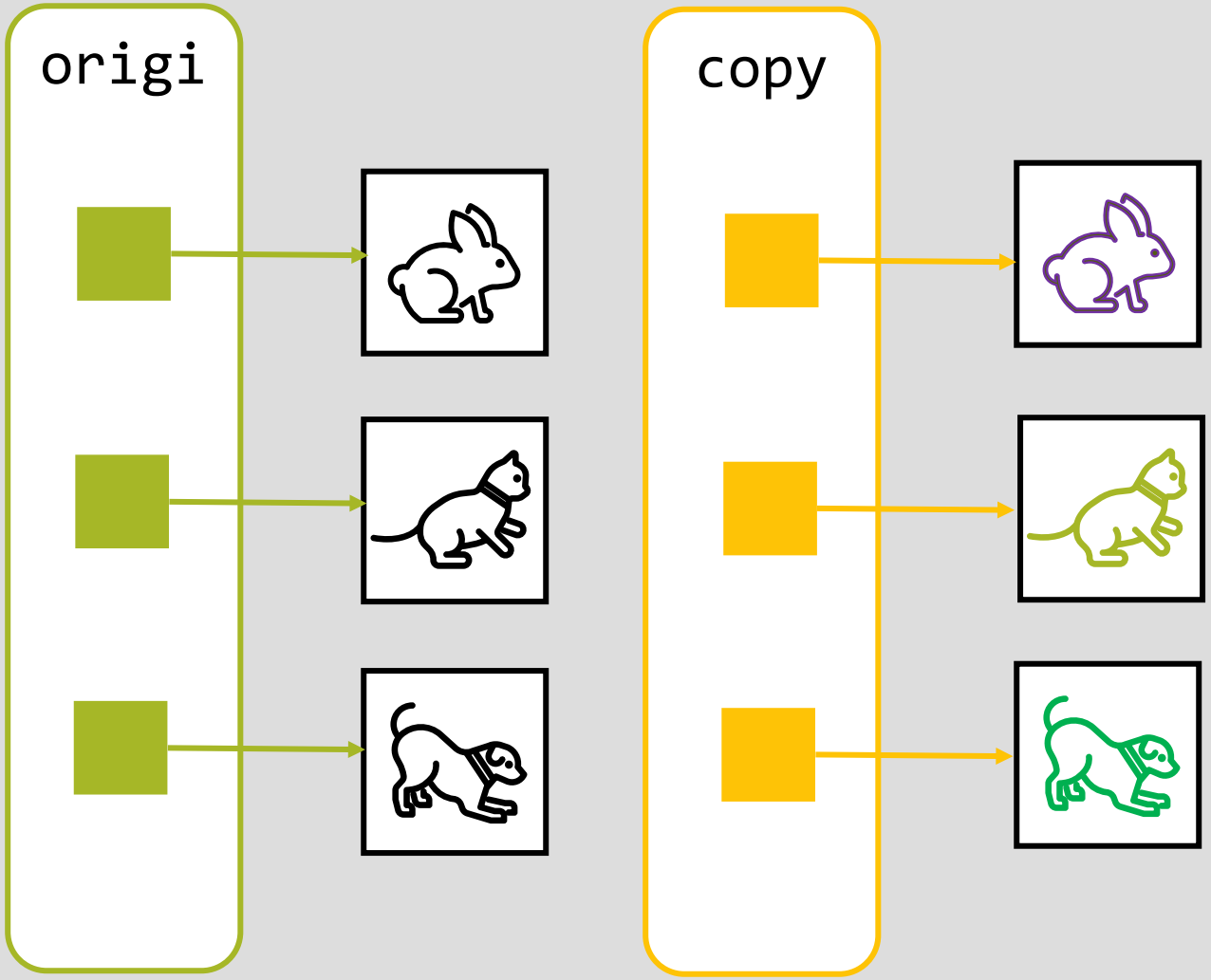
Shallow Clone vs Deep Clone

(if time permits)

Shallow Clone

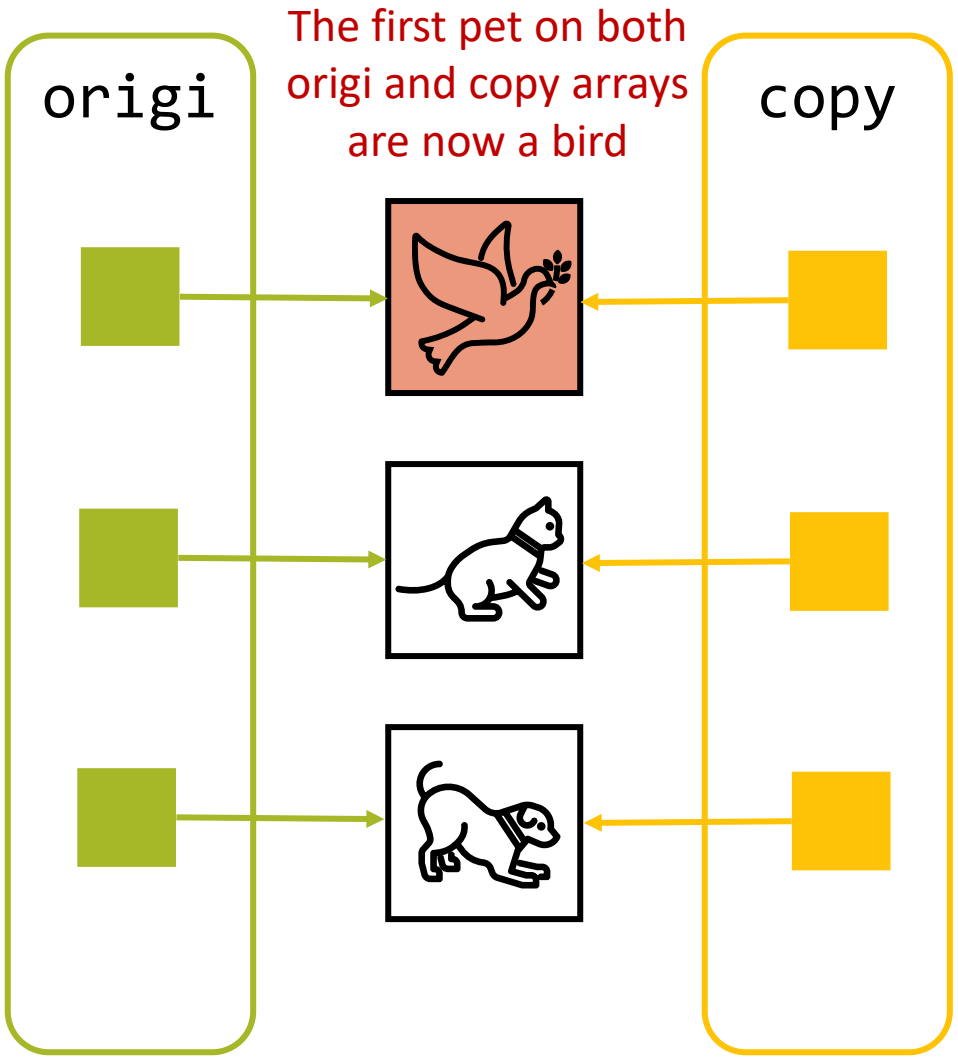


Deep Clone

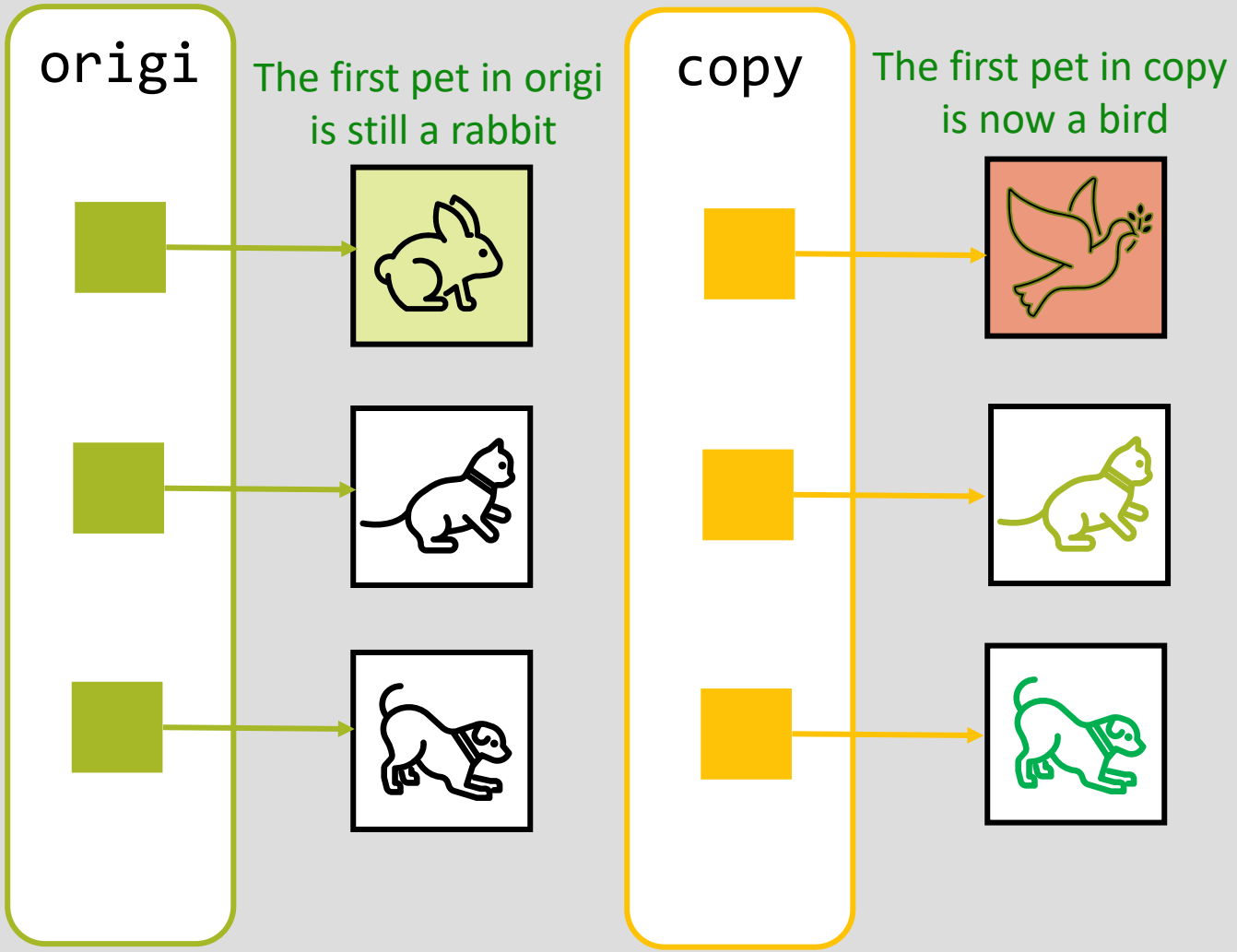


After we update the first pet in "copy" array to be a bird, what happened?

Shallow Clone



Deep Clone



```java
Animal[] pets1 = new Animal[3];
pets1[0] = new Animal("rabbit");
pets1[1] = new Animal("cat");
pets1[2] = new Animal("dog");

for(Animal a: pets1){
    System.out.println("pets1: " + a.toString());
}

Animal[] pets2 = new Animal[3];
/*********************************
 * Shallow Clone
 */
pets2 = pets1;

for(Animal a: pets2){
    System.out.println("pets2: " + a.toString());
}

// change animal at index 0
pets2[0].name = "bird";

System.out.println("pets1[0]:" + pets1[0] + ", " + pets1[0].name);
System.out.println("pets2[0]:" + pets2[0] + ", " + pets2[0].name);
```

```
pets1: Animal@7852e922
pets1: Animal@4e25154f
pets1: Animal@70dea4e
pets2: Animal@7852e922
pets2: Animal@4e25154f
pets2: Animal@70dea4e
pets1[0]:Animal@7852e922, bird
pets2[0]:Animal@7852e922, bird
```

```java
Animal[] pets1 = new Animal[3];
pets1[0] = new Animal("rabbit");
pets1[1] = new Animal("cat");
pets1[2] = new Animal("dog");

for(Animal a: pets1){
    System.out.println("pets1: " + a.toString());
}

Animal[] pets2 = new Animal[3];

/*********************************
 * Shallow Clone
 */
for(int i = 0; i < 3; i++){
    pets2[i] = pets1[i];
}

for(Animal a: pets2){
    System.out.println("pets2: " + a.toString());
}

// change animal at index 0
pets2[0].name = "bird";

System.out.println("pets1[0]:" + pets1[0] + ", " + pets1[0].name);
System.out.println("pets2[0]:" + pets2[0] + ", " + pets2[0].name);

}
```

Shallow Clone

Deep Clone

```
Animal[] pets1 = new Animal[3];
pets1[0] = new Animal("rabbit");
pets1[1] = new Animal("cat");
pets1[2] = new Animal("dog");

for(Animal a: pets1){
    System.out.println("pets1: " + a.toString());
}

Animal[] pets2 = new Animal[3];

/*********************************
 * Deep Clone
 */
for(int i = 0; i < 3; i++){
    pets2[i] = new Animal(pets1[i].name);
}

for(Animal a: pets2){
    System.out.println("pets2: " + a.toString());
}

// change animal at index 0
pets2[0].name = "bird";

System.out.println("pets1[0]:" + pets1[0] + ", " + pets1[0].name);
System.out.println("pets2[0]:" + pets2[0] + ", " + pets2[0].name);
```

```
pets1: Animal@7852e922
pets1: Animal@4e25154f
pets1: Animal@70dea4e
pets2: Animal@5c647e05
pets2: Animal@33909752
pets2: Animal@55f96302
pets1[0]:Animal@7852e922, rabbit
pets2[0]:Animal@5c647e05, bird
```