Object-Oriented Programming

Arrays
Generics

Computer Science and Technology
United International College

Outline

- Arrays of primitive values.
- Arrays of objects.
- Enhanced **`for`** loops.
- Java's **`ArrayList`** class.
- Generics (parametric polymorphism).
- Generic **`ArrayList`**.

Arrays of primitive types

- "An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed." (Oracle documentation)
- Since a Java array is an object, it must be created separately from the variable referencing the array, using the `new` operator.
- The type of the variable must match the type of the array but there is no need to specify the length of the array in the type of the variable.

```
int[] a;            // Variable.
a = new int[3]; // Array object.
```

Arrays of primitive types

- If the elements of the array are of a primitive type (**`int`**, **`double`**, etc.; not a class type) then creating the array object is all you need to do.
- Array elements can then be read / written using the same `[]` array notation as in C.
- Array indexes start at `0`, just like in C.
- Even though the number of elements of an array is fixed, it can be decided dynamically at runtime:

```
int n = ...; int[] a = new int[n];
```

- The number of elements in the array is stored in a public instance variable of the array object called **`length`**.

Arrays of primitive types

```
public class Test {
    public static void main(String[] args) {
        int[] a;          // Variable.
        a = new int[3]; // Array object.
        System.out.println("length: " + a.length);
        for(int i = 0; i < a.length; i++) {
            a[i] = 2 * i;
            System.out.println("a[" + i + "] = " + a[i]);
        }
    }
}
```

Arrays of objects

- If the elements of the array are objects then creating the array object itself is not enough!
- You must also create each element of the array one by one using the `new` operator (inside a loop).
- Java does not do this automatically for you because it cannot guess how you want to create the elements of the array (which constructor to use, etc.)
- Everything else works as usual.

Arrays of objects

```
public class Student {
    private String name;
    public Student(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Arrays of objects

```
public class Test {
    public static void main(String[] args) {
        Student[] a;          // Variable.
        a = new Student[3]; // Array object.
        System.out.println("length: " + a.length);
        // Creating all the array elements one by one:
        for(int i = 0; i < a.length; i++) {
            a[i] = new Student("Student " + i);
        }
        for(int i = 0; i < a.length; i++) {
            // a[i] is of type Student.
            System.out.println("name: " + a[i].getName());
        }
    }
}
```

Enhanced **for** loop

In addition to normal loops (**for**, **while**, **do-while**), Java provides an "enhanced **for** loop" which makes array processing easier:

```
for(Student s: a) {
    System.out.println("name: " + s.getName());
}
```

Internally it works like this:

```
for(int i = 0; i < a.length; i++) {
    Student s = a[i];
    System.out.println("name: " + s.getName());
}
```

Enhanced **for** loop

- The general shape is:

```
for(elementType varName: arrayName) {
    … varName …
}
```

- Internally the Java compiler automatically transforms such a loop into a normal `for` loop.
- `varName` is then a synonym (another name) for `arrayName[i]`.
- `elementType` can be a primitive type or a class type, both work the same way.

Enhanced `for` loop

Advantages:

- It is easier to write: `for(Student s: a) { … }`
- It is easier to read: `for` each `Student` `s` in the array `a`, do something…
- You do not have to worry about the details of the indexing (initializing an index variable, comparing the index with the length of the array, incrementing the index).
- So less opportunities for indexing errors.

Enhanced `for` loop

Disadvantages:

- Because **`varName`** is only a synonym for **`arrayName[i]`**, and is not **`arrayName[i]`** itself, modifying **`varName`** does not modify the array object!
- The array elements are always all accessed one by one in order of increasing index (from **`0`** to **`length - 1`**) and there is no way to change that.

So if you want to modify the content of the array or access the array element in non-increasing order or skip some array elements then you cannot use an enhanced **`for`** loop, you must use a normal loop.

Enhanced `for` loop

```
public class Test {
    public static void main(String[] args) {
        Student[] a;          // Variable.
        a = new Student[3]; // Array object.
        // a[i] is modified so use a normal loop:
        for(int i = 0; i < a.length; i++) {
            a[i] = new Student("Student " + i);
        }
        // a[i] is not modified so use an enhanced loop:
        for(Student s: a) {
            System.out.println("name: " + s.getName());
        }
    }
}
```

Enhanced `for` loop

- Note: for an array of objects (not an array of primitive types) there is a difference between the array object itself and the element objects stored in the array.
- It is not possible to use an enhanced `for` loop to modify the array object.
- It is possible to use an enhanced `for` loop to modify the element objects!
- Do not confuse the array object with its element objects!

Enhanced `for` loop

```
public class Test {
    public static void main(String[] args) {
        Student[] a;          // Variable.
        a = new Student[3]; // Array object.
        // a[i] is modified so use a normal loop.
        for(int i = 0; i < a.length; i++) {
            a[i] = new Student("Student " + i);
        }
        for(Student s: a) { // Works as expected.
            s.setName(s.getName() + " new");
        }
        for(Student s: a) {
            System.out.println("name: " + s.getName());
        }
    }
}
```

Java's `ArrayList`

- `ArrayList` is a class provided by Java.
- Just like an array, an arraylist is an object that can contain other objects.
- Just like an array, you can access elements of the arraylist using an index that starts at `0`.
- Just like a list, you can grow or shrink the size of the arraylist dynamically by adding or removing elements.
  - The initial size of an arraylist is zero.
- Very convenient to use.

Java's `ArrayList`

- By default the type of the elements of an arraylist is **`Object`**.
  - This allows an arraylist to contain any kind of object.
  - A downcast is then usually required when reading an element from an arraylist.
- **`add`**, **`get`**, and **`set`** methods must be used to add, read, and write elements of the arraylist: the usual array notation does not work.

Java's ArrayList

```
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        ArrayList a;            // Variable.
        a = new ArrayList(); // ArrayList object.
        // Loop up to 3 because a.size() is 0 initially.
        for(int i = 0; i < 3; i++) {
            a.add(new Student("Student " + i)); // Upcast.
        }
        for(int i = 0; i < a.size(); i++) {
            Student s = (Student)a.get(i); // Downcast.
            System.out.println("name: " + s.getName());
        }
    }
}
```

Java's `ArrayList`

- Arraylists are mostly used to store objects.
- Arraylists can also be used with primitive values:
  - Java then automatically converts the primitive value into an equivalent object: `int` becomes `Integer`, `double` becomes `Double`, `boolean` becomes `Boolean`, etc.
    - These classes are provided by Java.
    - This automatic conversion is called boxing.
  - The object equivalent to the primitive value is then stored in the arraylist.
  - Later when taking the object out of the arraylist (and doing a downcast), Java can automatically unbox the object back into the original primitive value.

Java's ArrayList

```
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        ArrayList a;          // Variable.
        a = new ArrayList(); // ArrayList object.
        // Loop up to 3 because a.size() is 0 initially.
        for(int i = 0; i < 3; i++) {
            // Box int into Integer and upcast Integer into Object.
            a.add(i);
        }
        for(int i = 0; i < a.size(); i++) {
            // Downcast Object into Integer and unbox Integer into int.
            int j = (int)a.get(i);
            // This work too:
            //int j = (Integer)a.get(i);
            System.out.println("value: " + j);
        }
    }
}
```

Java's `ArrayList`

- Enhanced **`for`** loops work with arraylists too.
- But you still need to do the downcast from **`Object`** back into the original type of the elements.
- Just like for array objects, do not try to modify an arraylist object from inside an enhanced **`for`** loop that loops over the same arraylist!
  - The Java compiler will allow it.
  - The loop will probably not work the way you want!

Java's ArrayList

```
import java.util.ArrayList;
public class Test {
   public static void main(String[] args) {
      ArrayList a;             // Variable.
      a = new ArrayList(); // ArrayList object.
      // Loop up to 3 because a.size() is 0 initially.
      for(int i = 0; i < 3; i++) {
         a.add(new Student("Student " + i)); // Upcast.
      }
      for(Object o: a) {
         Student s = (Student)o; // Downcast.
         System.out.println("name: " + s.getName());
      }
   }
}
```

Java's `ArrayList`

- Wouldn't it be nice to not have to write these downcasts all the time when reading an element from an arraylist?
- And the JVM checks all the downcasts at runtime so the downcasts slow down the program too.

- If only we could specify explicitly the type of the elements of the arraylist…
- … and get rid of all the downcasts …
- … and let the Java compiler do all the type checks at compile time.

Generics

- Generics are a way to parameterize a class over a type.
  - A method can take a value as argument.
  - Similarly, a generic class can take a type as argument.
- Also called parametric polymorphism.
  - Java's third and last kind of polymorphism, after ad-hoc polymorphism (overloading) and subtyping polymorphism (from inheritance and interface implementation).

Generics

Then:

- We don't need downcasts anymore when reading elements from an object such as an arraylist.
- All type errors can be found at compile time.

Generics are also useful when we have two classes that have exactly the same code but with different types.

- Example: a `Box` class.

Generics

```
public class Box {
    private int data;
    public Box(int data) {
        this.data = data;
    }
    public int getData() {
        return data;
    }
    public void setData(int data) {
        this.data = data;
    }
    public static void main(String[] args) {
        Box b = new Box(1);
        System.out.println(b.getData() == 1);
        b.setData(2);
        System.out.println(b.getData() == 2);
    }
}
```

Generics

```
public class Box {
    private boolean data;
    public Box(boolean data) {
        this.data = data;
    }
    public boolean getData() {
        return data;
    }
    public void setData(boolean data) {
        this.data = data;
    }
    public static void main(String[] args) {
        Box b = new Box(true);
        System.out.println(b.getData() == true);
        b.setData(false);
        System.out.println(b.getData() == false);
    }
}
```

Generics

- The two `Box` classes are exactly the same, except for:
  - The types which are different.
  - The test values which are different (they must be, since the types are different!)
- Since Java does not allow two classes to have the same name, we must also use two different class names (such as `IntBox` and `BoolBox`).
- Software engineering: code duplication is bad.

What if we use the `Object` type to try to solve the problem?

Generics

```
public class Box {
    private Object data;
    public Box(Object data) {
        this.data = data;
    }
    public Object getData() {
        return data;
    }
    public void setData(Object data) {
        this.data = data;
    }
    ...
```

Generics

```
    ...
    public static void main(String[] args) {
        Box b1 = new Box(1);
        System.out.println((int)b1.getData() == 1);
        b1.setData(2);
        System.out.println((int)b1.getData() == 2);
        Box b2 = new Box(true);
        System.out.println((boolean)b2.getData() == true);
        b2.setData(false);
        System.out.println((boolean)b2.getData() == false);
    }
}
```

Generics

- Using `Object` works but then we have downcasts everywhere again, just like when we use an arraylist!

Instead:

- Using generics, the type used in the code can become a type parameter of the class: `T` (or any other name you like).
- The actual type is then only specified when you use the class.

Generics

```
public class Box<T> {
    private T data;
    public Box(T data) {
        this.data = data;
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data = data;
    }
    ...
```

Generics

```
...
public static void main(String[] args) {
    Box<Integer> b1 = new Box<Integer>(1);
    System.out.println(b1.getData() == 1);
    b1.setData(2);
    System.out.println(b1.getData() == 2);
    Box<Boolean> b2 = new Box<Boolean>(true);
    System.out.println(b2.getData() == true);
    b2.setData(false);
    System.out.println(b2.getData() == false);
  }
}
```

Generics

- `class Box<T>` means that the `Box` class is generic and it is using the type parameter `T` as the name for some unknown type (to be specified later).
- Instance variables and methods can then use `T` just like any other type, even though we do not know what `T` is!
- It is only later when we use the `Box` class that we specify what `T` is:

```
Box<Integer> b1 = new Box<Integer>(1);
...
Box<Boolean> b2 = new Box<Boolean>(true);
```

Generics

- We can now use the same `Box` class with any type `T` that we want!
- There is no need for downcasts anymore, Java knows exactly what kind of value is stored in which box, based on the type of the box itself.
- All types can be checked at compile time.
  - So errors in your code are detected before you ship your software to your customers!
- The code runs faster too.
- And there is no code duplication.

Life is beautiful!

Java's generic `ArrayList`

- Java's `ArrayList` class is a generic class too.
- Therefore we can use `ArrayList` with any type we want: we just have to specify which type we want for the arraylist's elements when using the `ArrayList` type.
- There is no problem using an enhanced `for` loop with generics either.
- So our old code that was using an arraylist with elements of type `Object` plus downcasts:

Java's generic ArrayList

```
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        ArrayList a;              // Variable.
        a = new ArrayList(); // ArrayList object.
        // Loop up to 3 because a.size() is still 0.
        for(int i = 0; i < 3; i++) {
            a.add(new Student("Student " + i)); // Upcast.
        }
        for(Object o: a) {
            Student s = (Student)o; // Downcast.
            System.out.println("name: " + s.getName());
        }
    }
}
```

now becomes:

Java's generic `ArrayList`

```
import java.util.ArrayList;
public class Test {
    public static void main(String[] args) {
        ArrayList<Student> a;              // Variable.
        a = new ArrayList<Student>(); // ArrayList object.
        // Loop up to 3 because a.size() is still 0.
        for(int i = 0; i < 3; i++) {
            a.add(new Student("Student " + i));
        }
        for(Student s: a) {
            System.out.println("name: " + s.getName());
        }
    }
}
```

and the downcasts are gone.

Generics

- Many of Java's classes are generic too (lists, queues, trees, hash maps, etc.) to make you life more beautiful.
- Generic classes can take more than one type parameter.
  - Example: `class HashMap<K,V> { … }`
- It is possible to restrict a generic class to work with only some types, instead of all types.
  - Example: `class Box<T extends Animal> { … }`
  - Only objects from the class **`Animal`** and its subclasses (**`Cat`**, **`Dog`**, etc.) can then be put inside a **`Box`** object.
  - **`T`** is then called a bounded type parameter.

Generics

- Interfaces can be generic too.
  - Example: any class implementing the **`interface Iterable<T>`** can be used with an enhanced **`for`** loop.
  - A generic interface can then be implemented by a generic class: **`public class Rabbit<T> implements Edible<T> { … }`**
- Generics have many many more features available, this is only a quick introduction!

[Generics Tutorial (Oracle web site)]

Summary

- Arrays of primitive values.
- Arrays of objects.
- Enhanced `for` loops.
- Java's `ArrayList` class.
- Generics (parametric polymorphism).
- Generic `ArrayList`.