CSC207 - Arrays, Inheritance, Generic Types

Ilir Dema

Summer 2016

What is an array?

An **array** is an arrangement of elements of the same type at equally spaced addresses in computer memory.

Algonquin Radio Observatory

`http://www.thothx.ca`

Arrays in Java

- An array in Java is not a primitive; it is an object. We can access its length through its `length` property.
- To declare an array we use the following syntax:

  ```
  DataType[] arrayName;
  ```

- Remember: Declaring a variable does not create any new objects.
- An array can be initialized as follows:

  ```
  arrayName = new DataType[arrayLength];
  ```

- One can define multidimensional arrays as arrays of arrays.

Java Collection Framework and Generics

- Arrays are a very simple data structure.
- Often is useful to have implementations of certain Abstract Data Types (ADT).
- Java's Colletion Framework provides access to different ADTs.
- It is desirable that the implementation of the ADTs be independent of the data type stored in the chosen structure.
- This can be achieved through so called generics - a way of extending static typing to classes when the exact type of data the classes will operate on is unknown.
- For example, we may be interested to create Lists of Strings, Points (recall example from last lecture), etc.
- A type enclosed within angle brackets, for example `ArrayList<T>` means the programmer should replace `T` with the desired data type.
- Example: `ArrayList<Point> polygon = new ArrayList<Point>();`

Inheritance Hierarchy in Java

- All classes form a tree called inheritance hierarchy with `Object` at the root.
- Class `Object` does not have a parent. All other Java classes have a single parent. Multiple inheritance in Java is not allowed.
- If a class has no parent declared, it is a child of class `Object`.
- A parent class can have multiple child classes.
- Class `Object` guarantees that every class inherits the following methods:

  `Object()`, `clone()` : Creational methods.
  `equals(Object)`, `hashCode()` : Test equality, compute object hash.
  `toString()`, `finalize()`, `getClass()` : convert instance to string, handle the destruction of the object, return the class of the current instance.
  Other methods (synchronizing methods).

Inheritance

- Inheritance allows one class to inherit the data and the methods of another class.
- In a subclass, `super` refers to the part of the object defined by the parent class.
  - Use `super.‘‘attribute’’` to refer an attribute (data member of method) in the parent class.
  - Use `super(‘‘arguments’’)` to call a constructor defined in the parent class.
- If the constructor of the parent class is intended to be called, the `super(‘‘arguments’’)` must be the first line of code of the constructor.
  - Otherwise the default (no argument) constructor in the parent class is called.

Multipart objects

Suppose class `Child` extends class `Parent`.

An instance of `Child` has

- a `Child` part, with all data members and methods of `Child`
- a `Parent` part, with all the data members and methods of `Parent`
- a `GrandParent` part, ...etc., all the way up to `Object`.

An instance of `Child` can be used anywhere that a `Parent` is legal.

- But not the other way around.

Name Lookup

A subclass can resuse a name already used for an inherited data member or method.

Example: class `Player` could have a data member `name` and so could class `Wizard`. Or they could both have a method with signature `fight(Player)`.

When we construct

```
x = new Wizard("Jenny",500,400,new
String[]{"abracadabra"});
```

the object has a `Player` part and a `Wizard` part.

If we say `x.healthPoints` or `x.fight`, we need to know which one we'll get!

In other words, we need to know how Java will look up the name `healthPoints` or `fight` inside a `Wizard` object.

Name lookup rules

For a method call: `expression.method(arguments)`

- Java looks for method in the most specific, or bottom-most part of the object referred to by expression.
- It is is not defined there, Java looks "upward" until it is found (else it is an error).

For a reference to an instance variable: `expression.variable`

- Java determines the type of the expression, and looks in that box.
- It is is not defined there, Java looks "upward" until it is found (else it is an error).

Shadowing and Overriding

Suppose class `A` and its subclass `AChild` each have an instance variable `x` and an instance method `m`.

`A`'s `m` is **overriden** by `AChild`'s `m`.

- This is often a good idea. We often want to specialize behaviour in a subclass.

`A`'s `x` is **shadowed** by `AChild`'s `x`.

- This is confusing and rarely a good idea.

If a method must not be overriden in a descendant, declare it `final`.

Overriding `equals()`

The `equals()` method should return `true` if the two objects can be considered equal and `false` otherwise.

Of course, what data is considered equal is up to each individual class to define.

The `equals()` method's signature must be:

- `public boolean equals(Object)`

The `equals()` method's Javadoc declares that it must be:

- Reflexive : an object must always be equal to itself; i.e., `a.equals(a)`
- Symmetric : if two objects are equal, then they should be equal in both directions; i.e., if `a.equals(b)`, then `b.equals(a)`.
- Transitive: if an object is equal to two others, then they must both be equal; i.e., if `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`.
- Non-null: an object can never be equal to null; i.e., `a.equals(null)` is always false.

Overriding `hashCode()`

The hash code is just an int calculated from the instance data.

If two instances are considered equal by `equals()`, then they must have the same hash code.

Therefore, the hash code can only be computed on those fields compared in the `equals()` method.

Casting

- Casting it's you telling the compiler that an Object of type A is actually of more specific type B, and thus gaining access to all the methods on B that you wouldn't have had otherwise.
- You can also perform casting with primitive data types e.g. casting a long variable into an int, casting a double variable into a float, or casting an int variable into char, short and byte data type. This is known as down-casting in Java.