CMSC 132

Intro to Object Oriented Programming II

Ekesh Kumar
Prof. Nelson Padua-Perez • Fall 2019 • University of Maryland
https://www.cs.umd.edu/class/fall2019/cmsc132/

Last Revision: September 15, 2019

Contents

1 Tuesday, August 27, 2019

This class is CMSC132: Object Oriented Programming II. We will be covering modern software development techniques, various algorithms and data structures, and more. This course is taught in Java 11, but the techniques taught in this course carry over to several other programming languages.

Logistics

- All lectures are recorded and posted on Panopto.
- No collaboration on projects.
- No pop quizzes.
- We will be using the Eclipse IDE.
- Projects, office hours, lecture videos, and most other resources can be accessed from the class webpage: https://www.cs.umd.edu/class/fall2019/cmsc132/.
- Projects are due at 11:30 p.m. on the specified day. They can, however, be submitted up to 24 hours late with a 12% penalty.
- The grade breakdown is 26% for Projects, 16% for Quizzes, Exercises, and Lab Work, 8% for Midterm 1, 11% for Midterm 2, 11% for Midterm 3, and 28% for the Final Exam.

Abstraction and Encapsulation

There are important two techniques used in Object-Oriented Programming that we need to become familiar with: abstraction and encapsulation.

Abstraction is a technique in which we provide a very high-level model of activity or data. Small details about the model's functionality are not specified to the user. A good example . Abstraction can further be divided into two sub-categories, which are described below:

1. **Procedural abstraction** is a type of abstraction in which the user is aware of what actions are being performed, but they are not told how the action is performed. For example, suppose we would like to support a list of numbers. There are many algorithms that can do this for us. Under procedural abstraction, we would know that our end result is a sorted list of numbers, but we wouldn't know which algorithm is being used.
2. **Data abstraction** is a type of abstraction in which various data objects are known to the user, but how they are represented or implemented is not known to the user. An example of data abstraction is shown by representing a list of people. While the user would know that they have a list of people, they wouldn't know how the list is being represented (for example, we could use an array, an ArrayList, or any other data structure).

An **abstract data type** (ADT) is an entity that has values and operations. More formally, an abstract data type is an implementation of interfaces (a set of methods). Note that it is "abstract" because it does not provide any details surrounding how these various operations are implemented.

An example of an abstract data type is a queue, which supports the operation of inserting items to the end of the queue as well as the operation of retrieving items from the front of the queue. Note, again, that we are not concerned with how these operations should be performed internally.

Finally, **encapsulation** is a design technique that calls for hiding implementation details while providing an interface (a set of methods) for data access. A familiar example of encapsulation is shown through the ArrayList in Java. The ArrayList provides various methods that are accessible to us, such as the `.add()` and `.at()` methods. We aren't concerned with how they are implemented internally.

The Java Programming Language

Different programming languages provide varying levels of support for object-oriented programming. In particular, Java and C++ allow us to easily perform object-oriented programming. How does Java does this?

- Java provides us with **interfaces**, which allows us to specify a set of methods that another class must implement. This allows us to easily express an abstract data type since we can specify the operations and data that an entity must have. Interfaces follow an "is-a" relationship, meaning that a class that implements an interface is what the interface specifies. As an example, consider an animal interface. If an elephant implements the animal interface, we can perform tasks that are meant for animals with elephants (such as passing an elephant into a function that accepts animals).
- Java provides us with **classes**, which can be used as blueprints for other classes. Classes can extend other classes, which makes them **subclasses**. These subclasses inherit functions from the original class, and this also defines an "is-a" relationship.

Here are some key points that we should remember about interfaces:

- An interface cannot be instantiated. So, if we have an interface called `animal`, typing `Animal a = new Animal()` **would not compile**.
- An interface can contain many public members, including static final constants, abstract methods (which have no body), default methods (with a body), static methods, and static nested types.

Let's look at an example of an interface:

```
import java.util.*;

public interface Animal {

    public void feed(String food);
    public int getAge();
    public boolean manBestFriend();

    default void grow() {
        System.out.println("I grow");
    }
}
```

Here, we've created an interface named `Animal`, which other classes can implement. Any class that implements this class will need to also implement the functions `feed`, `getAge()` and `manBestFriend()` with the same return values and parameters specified. The `grow()` function will be associated with any class that implements this interface.

Now let's look at another class that implements the `Animal` interface:

```
package animalExample;

public class Dog implements Animal {
  private String name;
```

```
   private int age;

   public Dog(String name, int age) {
      this.name = name;
      this.age = age;

   }

   public void feed(String food) {
      System.out.println("Feeding " + name + " with " + food);
   }

   public int getAge() {
      return age;
   }

   public boolean manBestFriend() {
      return true;
   }

   public void bark() {
      System.out.println(name + " is barking");
   }
}
```

Here are the key things that we should note:

- The `Dog` class implements the `Animal` class. Thus, it is **required** to implement all of the methods specified by `Animal` (if we were to comment out even one function, the code wouldn't compile).
- The `Dog` class is allowed to add its own methods, like `bark()`. Likewise, the `Dog` class is allowed to create its own variables, like `name` and `age`.

Here's another class that implements the `Animal` interface:

```
package animalExample;

public class Piranha implements Animal {
   private int age;

   public Piranha(int age) {
      this.age = age;
   }

   public void feed(String food) {
      System.out.println("Piranha is eating " + food);
   }

   public int getAge() {
      return age;
   }

   public boolean manBestFriend() {
      return false;
   }

   public void grow() {
```

```
    System.out.println("I grow like a fish");
  }
}
```

Here are some more things that we should take careful note of:

- More than one class can implement an interface. In this example, both `Dog` and `Piranha` implement `Animal`.
- We can rewrite methods that are provided to us. More specifically, `Piranha` implements `Animal`, but it rewrites the `grow()` function that was originally provided to it.

Recall that interfaces define an "is-a" relationship. Since `Piranha` and `Dog` implement `Animal`, we can now treat them as an `Animal`. A good way to verify the "is-a" relationship is the `instanceof` keyword. In Java, `instanceof` allows us to check whether one object is an instance of a specified type (class, subclass, or interface).

To demonstrate, suppose we declare a piranha variable `p`. We could write a conditional such as `if (p instanceof Animal) { /* Code to be executed */}`, in order to make sure that `p` is an instance of `Animal`. Why would we ever need to do this? This is particularly useful when we're casting. If we attempt to cast an object to a subclass of which it is not an instance of, Java throws a **ClassCastException**. We can prevent this exception by putting the relevant code in a conditional checking the instance of what we're casting.

2 Wednesday, August 30, 2019

Collections

The **Java Collections Framework** is a set of methods and classes that provide support for **collections**, which are objects that group multiple elements into one unit. There is a collection interface that defines a set of methods that certain classes must have. It turns out that ArrayList is in the collections framework, and it implements the set of required methods (such as `.add()`, `.clear()`, `.contains()`, and more).

Consider the following code segment:

```
package miscellaneous;

import java.util.Collection;
import java.util.ArrayList;
import java.util.LinkedList;

public class CollectionExample {

   public static Collection<Integer> defineElements(boolean arrayFlag, int numberOfValues, int
       maxValue) {
      Collection<Integer> elements;

      if (arrayFlag) {
         elements = new ArrayList<Integer>();
      } else {
         elements = new LinkedList<Integer>();
      }

      for (int j = 0; j < numberOfValues; j++) {
         elements.add((int) (Math.random() * maxValue + 1));
      }

      return elements;
   }

   public static void displayValues(Collection<Integer> data) {
      if (data.isEmpty()) {
         System.out.println("Empty Collection");
      } else {
         for (Integer value : data)
            System.out.print(value + " ");
      }
   }

   public static void main(String[] args) {
      Collection<Integer> firstCollection = defineElements(true, 5, 10);

      System.out.println("First Collection:");
      displayValues(firstCollection);

      System.out.println("\nSecond Collection:");
      Collection<Integer> secondCollection = defineElements(false, 5, 10);
      displayValues(secondCollection);

      /* Example of methods defined by the Collection interface */
      Collection<Integer> union = new ArrayList<Integer>();
```

```
      /* Combining elements */
      union.addAll(firstCollection);
      union.addAll(secondCollection);
      System.out.println("\nUnion");
      displayValues(union);

      /* Checking if elements from one collection are present in antoher */
      if (union.containsAll(firstCollection))
         System.out.println("\nIncludes all elements of first collection");

      /* Removing elements in union present in second collection */
      union.removeAll(secondCollection);
      System.out.println("After removing");
      displayValues(union);

      /* Clearing the collection */
      union.clear();
      System.out.println();
      displayValues(union);
   }
}
```

This code segment is another demonstration of how we can utilize the fact that some objects are instances of other types. The `defineElements` takes in various parameters and returns either a LinkedList (another type list provided in the Collections Framework) or an ArrayList depending on the parameters provided. The reason why this is allowed is because of the "is-a" relationship between ArrayLists, LinkedLists, and Collections. More specifically, both ArrayLists and LinkedLists are collections. Consequently, since the return type of this function is a Collection, we are allowed to return any object that is an instance of Collection.

On the other hand, the `displayValues` method takes in a Collection type, and we're allowed to pass in any object that is an instance of Collection, as seen in the driver.

Collections are also really useful because they have built-in functions like `.shuffle()` (which scrambles the collection) and `.sort()`, which sorts the function.

Generics

We typically write, `ArrayList<Integer> = new ArrayList<Integer>()`, when we're instantiating an ArrayList of integers. Likewise, we could write, `ArrayList<Bananas> = new ArrayList<Bananas>()` if we had created a class called `Bananas`, or `ArrayList<String> = new ArrayList<String>()` if we wanted an ArrayList of strings. This is example of **generics**: we are allowed to pass in whatever type we want our ArrayList to store.

Why are generics useful? Firstly, generics help us reduce the amount of code we need to write. Instead of writing lots of code that gets executed depending on whether the user is using an integer or a string, we can use generics in order to reduce the amount of code needed. The `.add()`, `.remove()` and other methods used by ArrayList are valid no matter what type we are storing in our ArrayList. If we didn't have generics, we'd need to have.

Secondly, generics help us move some casting errors from runtime to compile time. Consider the following code segment:

```
class A { ... }
class B { ... }
List myL = new ArrayList(); // raw type
myL.add(new A()); // Add an object of type A
```

```
...
B b = (B) myL.get(0);
```

Here, we've declared two classes, and we've created a List called `myL`. In this example, we have not used any generics (this is called a **raw type**) since we haven't specified the type that this list will store. Thus, Java will permit us to add whatever we want to this list. This can lead to issues if we don't remember what type is stored at each index (in the above example, for instance, retrieved the first element by casting it as `B` when it is actually of type `A`.). This issue won't be seen until runtime. Using generics, however, would move this error to compile time.

Here is the same code segment but with generics:

```
class A { ... }
class B { ... }
List<A> myL = new ArrayList<A>();
myL.add(new A());
A a = myL.get(0);
...
B b = (B) myL.get(0); // Compile-time error
```

Now that we've used generics, we'll find the casting error during compile time.

So far, we've only seen how collections use generics. How do we use our own generics? We can parametrize classes, interfaces, and methods using `<X>` notation. For example, we could write, `public class foo<X, Y, Z> { ... }`, to define a class called `foo` that takes in three types.

The Iterable Interface

The **Iterable Interface** defines a method called `Iterator`, which returns an iterator that allows us to traverse a container (particularly lists). ArrayLists in Java implement the iterable interface.

The following example demonstrates how we can use iterators:

```
public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("a");
    myList.add("b");

    Iterator<String> it = myList.iterator();

    while (it.hasNext()) {
        System.out.println(it.next());
    }
}
```

This code segment will print out everything in our list `myList`.

However, it turns out that we don't actually need to handle iterators ourselves. We can use the **enhanced for loop** (also known as a **for each loo**), which does this for us. These types of for loops handle the iterator automatically.

Here's how we can use an enhanced for loop with our previous example:

```
public static void main(String[] args) {
    ArrayList<String> myList = new ArrayList<String>();
    myList.add("a");
    myList.add("b");
```

```
    for (String s : myList) {
        System.out.println(s);
    }
}
```

The for loop in this code segment can be read as, "for each string `s` in `myList`, print `s`."

3 Wednesday, September 4, 2019

The Comparable Interface

The **comparable interface** is used in order to provide an ordering to objects of a user-defined class. When a class implements this interface, it must implement the `.compareTo()` method, which returns a negative value if the current object is less than the object being compared to, zero if they're equal, and positive otherwise.

As an example, consider the following code segment:

```
package equalsComparable;
import java.util.ArrayList;

public class Example {

    public static void main(String[] args) {
        ArrayList<String> l = new ArrayList<String>();
        l.add("ice_cream");
        l.add("apple");
        l.add("raw_onion");
    }
}
```

If we were to print this ArrayList with `System.out.println(l)`, we'd get the output `[ice_cream, apple, raw_onion]`, which is the order in which we inserted the elements into the ArrayList.

But now, what if we wanted to sort our elements and then print them? Calling `Collections.sort(l)` and then printing would sort our items lexicographically, so our output would be `[apple, ice_cream, raw_onion]`. Why does this happen? Because the string class has a built-in `compareTo` method, which sorts lexicographically.

It turns out that we can define our own `compareTo()` method and sort our elements according to how we want. As mentioned previously, this can be done by implementing the `Comparable` interface.

Consider the following code segment, which illustrates how we can use the `Comparable` interface:

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

   public Student(String name, int id) {
      this.name = name;
      this.id = id;
   }

   public String toString() {
      return "Name: " + name + ", Id: " + id;
   }

   public int compareTo(Student other) {
      if (id < other.id) {
         return -1;
      } else if (id == other.id) {
         return 0;
      } else {
         return 1;
      }
   }
```

```
  public boolean equals(Object obj) {
     if (obj == this) {
        return true;
     } else if (!(obj instanceof Student)) {
        return false;
     } else {
        return compareTo((Student) obj) == 0;
     }
  }

  /* If we override equals we must have correct hashCode */
  public int hashCode() {
     return id;
  }
}
```

Note that the `Student` class implements the `Comparable` interface (whatever comes in the angled brackets `< ... >` specifies what we will be comparing). Consequently, the `compareTo` method is overriden so that we can specify that we want to sort the students based on their IDs (we retunr a negative value when the student's ID is less than the other students, zero when they're equal, and a positive value otherwise). Now, if we use `Collections.sort()` on an ArrayList of `Student` objects, we will sort based on ID.

An even simpler implementation of `compareTo` that would do just what we want would be to only have the statement `return id - other.id`.

Finally, we have a `hashCode` method, which will be discussed in-depth at a later time. But for now, it's important to know that, whenever the `equals` method is overriden, we must provide a `hashCode` method with the same header as shown in the code segment above. Java's enforcement of this "policy" is known as a **hash code contract**.

Annotations

In Java, **annotations** are used to help find runtime errors at compile time, and also provide additional information to others as to what you are trying to do. The general syntax for an annotation is `@[annotation name]`. Some of the common ones used by the compiler are listed below:

- `@Test` indicates that a test follows.
- `@Override` indicates that an overridden function follows.
- `@SuppressWarnings` indicates that the compiler should suppress warnings surrounding the code that follows.

What's an example in which annotations can help us?

Suppose we're overriding a function. If we make a mistake and write the function header for the overridden wrong, our compiler won't interpret this as an error (assuming everything else is right). But, if we were to put the `@Override` annotation before we override the function, our compiler will notice that we've got the wrong function header, and it will give us an error.

This is particularly helpful because it helps us identify otherwise latent bugs.

4 Friday, September 6, 2019

An Introduction to Inheritance

As mentioned briefly last class, **inheritance** is a technique that allows us to capitalize on some features that have already been implemented in a class. Ultimately, this allows us to reduce code duplication, and it also increases readability.

Suppose, for the sake of example, that we have a class `Shape`, and we want to implement several other related classes, like `Square`, `Rectangle`, and `Circle`. Since squares, rectangles, and circles are all examples of shapes, this is a clear example in which we can use inheritance. In computer science terminology, we would say that the `Square, Rectangle,` and `Circle` classes are **subclasses** or the `Shape` **superclass**. These subclasses **extend** their superclass.

In essence, by putting the variables and methods that are common to all shapes in the `Shape` class, we can avoid having to re-add them in every class that extends the `Shape` class (the subclasses can inherit what we want them to). We can subsequently define new instance variables and new methods that are specific to the shape we are implementing. For example, the subclasses might inherit a floating-point `area` field from the `Shape` class, but they must implement their `getArea()` methods in different ways.

The following classes, used to represent a University Database, clearly depict how some features of inheritance work. There are three classes that compose this program.

First, the `Person` class:

```
package university;

public class Person {
   private String name;
   private int id;

   public Person(String name, int id) {
      this.name = name;
      this.id = id;
   }

   public Person() {
      this("Unknown", 0);
   }

   public Person(Person person) {
      this(person.name, person.id);
   }

   public String getName() {
      return name;
   }

   public int getId() {
      return id;
   }

   public void setName(String name) {
      this.name = name;
   }

   public void setId(int id) {
```

```
      this.id = id;
   }

   public String toString() {
      return "[" + name + "] " + id;
   }

   public boolean equals(Object obj) {
      if (obj == this)
         return true;
      if (!(obj instanceof Person))
         return false;
      Person person = (Person) obj;

      return name.equals(person.name);
   }

   public int hashCode() {
      return id;
   }

   public static void main(String[] args) {
      Person p1 = new Person("Paul", 10);
      Person p2 = new Person("Mary", 20);
      Person p3 = new Person(p2);

      System.out.println(p1);
      System.out.println("Same?: " + p1.equals(p2));
      System.out.println("Same?: " + p2.equals(p3));
   }
}
```

The `Person` class, shown above, is a very general class used to represent an individual person. There are some fields, like `name` and `id`, which characterize a person attending the university we are representing. Also, there are a few different ways in which we can instantiate a `Person` object, as seen by the different constructors provided. Note that we have also override the `equals` method, which simply compares the names of the people. As we've learned, we can prevent some compile-time errors by adding the `@Override` annotation before this method. Finally, we have provided a `hashCode` method in order to satisfy Java's hash code contract (as mentioned before, this will be discussed in-depth at a later time).

Next, we have the `Student` class:

```
package university;

public class Student extends Person {
   private int admitYear;
   private double gpa;

   public Student(String name, int id, int admitYear, double gpa) {
      super(name, id); /* calls super class constructor */
      this.admitYear = admitYear;
      this.gpa = gpa;
   }

   /* What would happen if we remove the Person default constructor? */
   public Student() {
      super(); /* calls base case constructor (what if we remove it?) */
```

```
    admitYear = -1;
    gpa = 0.0;
  }

  public Student(Student s) {
    super(s); /* calls super class copy constructor */
    admitYear = s.admitYear;
    gpa = s.gpa;
  }

  public int getAdmitYear() {
    return admitYear;
  }

  public double getGpa() {
    return gpa;
  }

  public void setAdmitYear(int admitYear) {
    this.admitYear = admitYear;
  }

  public void setGpa(double gpa) {
    this.gpa = gpa;
  }

  public String toString() {
    /* Using super to call super class method */
    return super.toString() + " " + admitYear + " " + gpa;
  }

  public boolean equals(Object obj) {
    if (obj == this)
      return true;
    if (!(obj instanceof Student))
      return false;
    Student student = (Student) obj;

    /* Relying on Person equals; passing student */
    return super.equals(student) && admitYear == student.admitYear;
  }

  public static void main(String[] args) {
    Student bob = new Student("Bob", 457, 2004, 4.0);
    Student robert = new Student(bob);
    Student tom = new Student("Tom", 457, 2004, 4.0);

    System.out.println(bob);
    System.out.println("Same:" + bob.equals(robert));
    System.out.println("Same:" + tom.equals(robert));
  }
}
```

Firstly, we can observe that the `Student` class extends the `Person` class, meaning that it extends the public methods and fields that `Student` has.

The **super** class, when used in a method, calls the method with the same method header in the superclass

of the class `super` is being called in.

Now that we've already implemented various constructors in the `Person` class, it isn't necessary to re-implement them in the `Student` class since we're using inheritance. In each of our `Student` constructors, we just make a `super()` call to access the corresponding constructor in the parent `Person` class, which does what we want it to do.

Furthermore, we see that, in the `Student` constructor, we call `super(name, id)`, which calls the superclass's constructor. Whenever we're making `super()` call in a constructor, it is necessary for the `super()` call to be the first statement in a constructor.

In the default `Student()` constructor, which takes no parameters, it's fine to remove the `super()` call. Java will automatically add it for you, and it will call the default constructor of the superclass.

It's also important to note how Java can recognize which constructor to use from the parent class. For example, just `super()` will call the default constructor in the parent class, whereas `super(s)` will call the copy constructor in the parent class. It is particularly interesting to note how the superclass takes a `Person` object in its copy constructor, whereas the `Student` subclass takes a `Student` object. This is perfectly fine since every student "is-a" person.

If we didn't want the `Student` class to access some method of the `Person` class, we could just make that method private.

Finally, we have the `Faculty` class:

```
package university;

public class Faculty extends Person {
   private int hireYear; /* year when hired */

   public Faculty(String name, int id, int hireYear) {
      super(name, id);
      this.hireYear = hireYear;
   }

   public Faculty() {
      super();
      hireYear = -1;
   }

   public Faculty(Faculty faculty) {
      /* Why are we using get methods for the first two ? */
      this(faculty.getName(), faculty.getId(), faculty.hireYear);
   }

   int getHireYear() {
      return hireYear;
   }

   void setHireYear(int year) {
      hireYear = year;
   }

   public String toString() {
      return super.toString() + " " + hireYear;
   }

   public boolean equals(Object obj) {
      if (obj == this)
         return true;
```

```
    if (!(obj instanceof Faculty))
      return false;

    Faculty faculty = (Faculty) obj;

    /* Relying on Person equals; passing student */
    return super.equals(faculty) && hireYear == faculty.hireYear;
  }

  public static void main(String[] args) {
    Faculty john = new Faculty("John", 20, 2004);
    Faculty jack = new Faculty(john);
    Faculty mary = new Faculty("Mary", 101, 2010);

    System.out.println(mary);
    System.out.println("Same:" + john.equals(jack));
    System.out.println("Same:" + jack.equals(mary));

    System.out.println("Id: " + john.getId());
    System.out.println("HireYear: " + john.getHireYear());
  }
}
```

Similar to the `Student` class, the `Faculty` class extends the `Person` class and uses its constructors with `super()`. Also similar to the `Student` class, the `super()` call in the default `Faculty()` constructor is optional (Java will automatically add one if you don't have it).

Here are a few key points that you should remember about inheritance:

- The `extends` keyword is used to specify that a class is a subclass of another class. For example, `public class Student extends Person { .. }` indicates that `Student` is a subclass of `Person`.
- The `this` keyword is used to refer to the current class we're in.
- Subclasses inherit everything from their superclasses that isn't private.
- Superclasses and subclasses illustrate an "is-a" relationship — as seen by the copy constructor example, we can use the subclass objects wherever Java is expecting a superclass object without error.
- The `super()` call can be invoked when initializing a subclass object in order to use the superclass constructor.
- A `super()` call must be the first statement in your constructor.
- If you don't use a `super()` call, Java will automatically invoke the base class's default constructor. What happens if the base class doesn't have a default constructor? We get a compile-time error.

A consequence of the "is-a" relationship between the subclass and superclass is seen by the validity of various assignments. Going back to our university example, since a `Student` is a `Person`, if we declared a `Student` object `s` and a person object `p`, it would be fine to do something like, `p = s`. More specifically, it's fine to assign a person to a student since a student is a person. However, the other way around is not allowed.

Also, if we don't like a method that the superclass provides to us, we can override it in order to get a new one. This is seen in the `toString()` method in the university example.

5 Monday, September 9, 2019

The Protected Keyword

Recall that when a subclass extends a superclass, it cannot access any private fields in the superclass. What happens if we want to permit subclasses of a superclass to access various fields, but we also want to retain some level of privacy? This is where the **protected** keyword can help us. Like `public` and `private`, the `protected` keyword is used to specify what can access a variable.

When do we declare a variable to be `protected`? We should use the `protected` keyword when we need to access it from the enclosing class (the class it is in) subclasses that extend that class, and other classes in the same package (folder) as the enclosing class.

Essentially, declaring variables as `protected` allows us to limit the visibility of a variable, while still permitting subclasses to access that variable.

Early and Late Binding

Consider the following code segment:

```
Faculty carol = new Faculty("Carol Tuffteacher", 458, 1995);
Person p = carol;
System.out.println(p.toString());
```

Given that both the `Person` and `Faculty` classes implement a `toString()` method, which one should be used when executing the print statement on the third line? Should it be the `Faculty` object's `toString()` method? Or, should it be the `Person` object's `toString()` method? There are good arguments for either choice:

- One argument is that the variable `p` was initially declared to be of type person. Thus, we should use the Person's `toString()` method. This is known as **early (static) binding**. The name makes sense since we know early on (as soon as we've declared the object) what method will be used.
- The other argument says that the object to which `p` is referring to was created as a `Faculty` object, so we should use the `Faculty` object's `toString()` method. This is known as **late (dynamic) binding**. Again, this name is fitting since we don't figure out which method will be called when the person is initially declared.

There are pros and cons to both early binding and late binding. On one hand, early binding is more efficient since the decision of what set of methods should be used is determined at compile-time. On the other hand, late binding allows for more flexibility.

By default, **Java uses late-binding**, so in our example, the `Faculty`'s `toString()` method will be called. Essentially, late binding says that the method that is called is dependent on the object's actual type, and not the declared type of the referring variable.

Polymorphism

Java's use of late binding enables us to use a single reference variable to refer to objects of many different types. In the following code segment, we illustrate this fact by creating an array of various university people:

```
Person[] list = new Person[3];
list[0] = new Person("Col. Mustard", 10);
list[1] = new Student("Ms. Scarlet", 20, 1998, 3.2);
```

```
list[2] = new Faculty("Prof. Plum', 30, 1981);
for (int i = 0; i < list.length; i++) {
    System.out.println(list[i].toString());
}
```

The above code segment is perfectly valid, and it will use different `toString()` methods depending on type of the object it at each entry.

This is an example of **polymorphism**, which can more concretely be defined as a method of using a single symbol to represent multiple different types. Note that each of `list[1], list[2],` and `list[3]` are polymorphic variables since we can assign to them a `Person, Student`, or `Faculty` type without error.

getClass and instanceof

In Java, objects can obtain information about their types dynamically. This is done using the `getClass` and `instanceOf` methods that Java provides.

These methods do exactly what they sound like. The `getClass()` method returns the runtime class of an object. Internally, this is represented as a location in memory. It is also perfectly valid to compare the return-value of `getClass()`. For example, if we have two objects `b1` and `b2`, we could use the conditional `if (b1.getClass() == b2.getClass()) {...}` in order to check whether or not `b1` and `b2` are instances of the same class. This conditional will evaluate to "true" if both `x` and `y` are of the same type (i.e. both are Strings).

Now, consider the following code segment in which `getClass()` returns false:

```
Person bob = new Person(...);
Person ted = new Student(...);

if (bob.getClass() == ted.getClass()) { ... } // false (ted is really a Student).
```

The conditional inside of the `if` clause will evaluate to "false." Why? Because the `getClass()` method compares the runtime class of an object. Here, `ted` was initially declared to be of type `Person`, but it refers to a `Student` type. This example further illustrates the fact that Java uses late (dynamic) binding.

Next, we'll discuss the `instanceof` keyword.

The `instanceof` keyword is is used to determine whether one object is an instance of some other class. In terms of inheritance, Class A is an instance of Class B if Class A extends Class B. It's important to remember that `instanceof` is an **operator** in Java, not a method call. Thus, we do not invoke `.instanceof` on the object (there shouldn't be a period!).

The following example illustrates how both `getClass` and `instanceof` operate:

```
package university;

public class InstanceGetClass {
   public static void main(String[] args) {
      Person bobp = new Person("Bob", 1);
      Person teds = new Student("Ted", 2, 1990, 4.0);
      Person carolf = new Faculty("Carol", 3, 2016);

      /* Notice we are using Faculty variable rather than Person */
      Faculty drSmithf = new Faculty("DrSmith", 4, 2010);

      if (bobp.getClass() == teds.getClass()) {
         System.out.println("1. bob and ted associated with same getClass value");
```

```
    }

    if (bobp instanceof Person) {
       System.out.println("2. bob instance of Person");
    }

    if (teds instanceof Student) {
       System.out.println("3. ted instance of Student");
    }

    if (teds instanceof Person) {
       System.out.println("4. ted instance of Person");
    }

    if (bobp instanceof Student) {
       System.out.println("5. bob instance of Student");
    }

    if (carolf instanceof Person) {
       System.out.println("6. carol instance of Person");
    }

    if (carolf instanceof Student) {
       System.out.println("7. carol instance of Student");
    }

    /* Following will not compile (compare against previous one) */
    /*
     * if (drSmithf instanceof Student) {
     * System.out.println("drSmith instance of Student"); }
     */
  }
}
```

In this class, we're declaring three objects, `bobp, teds`, and `carolf` each of which have type `Person`. The variable `bobp` refers to a person, `teds` refers to a `Student`, and `carolf` refers a `Faculty` object. We also declare `drSmithf` to be of type `Faculty`, and `drSmithf` also refers to a `Faculty` object.

Now, which of these `if (...)` clauses will evaluate to "true," and which will evaluate to false?

- The conditional `bobp.getClass() == teds.getClass()` evaluates to false. Why? Because `bobp` is a `Person`, whereas `teds` is a `Student`. Thus, the first print statement isn't printed.
- The conditional `bobp instanceof Person` evaluates to true since `bobp` is a Person.
- The conditional `teds instanceof Student` evaluates to true since `teds` refers to a `Student`.
- `teds instanceof Person` evaluates to true since `teds` refers to a `Student`, and a `Student` extends the `Person` class.
- `bobp instanceof Student` is false since `bobp` has a `Person` object, and `Person` does not extend `Student`.
- `carolf instanceof Person` evaluates to true since the `Faculty` class extends `Person` class.
- `carolf instanceof Student` is false since `Faculty` does not extend `Student`.
- `drSmithf instanceof Student` is false since `drSmithf` is a `Faculty` object and also refers to a `Faculty` type. In fact, a `Faculty` object can never refer to a `Student` object, so the Java compiler recognizes this and issues a compile-time error (the conditional can never be true).

6 Wednesday, September 11, 2019

Upcasting and Downcasting

When casting in the context of inheritance, there are two different types of casting: upcasting and downcasting. What are the differences?

- **Upcasting** is when we cast a subtype to a supertype. For example, we could cast a `Student` object to a `Person` object in order to perform a task that operates on people.
- **Downcasting** is just going in the other direction: casting a supertype to a subtype.

Upcasting is always allowed. There is no problem in treating a subclass object as its superclass type due to the "is-a" relationship exhibited by the sub and superclasses. Upcasting can automatically be done by the compiler in some instances. For example, if we have a function that takes in a `Person`, we could pass in a `Student`, and the compiler would automatically cast the `Student` as a `Person`.

On the other hand, downcasting isn't always allowed.