Advanced Object Oriented Programming

Polymorphism: Inheritance and Methods

- Polymorphism is further demonstrated when the referenced object determines which method to execute.
- Methods in the superclass are available in its subclass. For example, the `toString` and `equals` method defined in the `Object` class is available in any classes.
- Methods in the superclass can be **overridden** in the subclass. For example, the default definition of the `toString` method in the `Object` class can be **overridden** to make the method more useful.
- When a method is called, Java would first look in the current class, then work its way up the hierarchy for the method (with the appropriate signature).
- Since the type of a reference variable need not to be the same as the object to which it refers, extra attention has to be put to determining which method is invoked.
  e.g. Suppose an `isOld` method has been written for `Person` and `Student` as follows:

```
class Person {
     // declaration of fields

     public boolean isOld () {
          return age > 65;
     }
}

class Student extends Person{
     // declaration of additional fields

     public boolean isOld () {
          return grade >= 11;
     }
}
```

And suppose the objects have been constructed as followed:

```
Person p = new Student();
```

The call `p.isOld()` is valid because `p` is of type `Person` and it has a method named `isOld`. However to determine which version of `isOld` to use, Java uses the object's type rather than the variable's type. The type of the variable `p` is `Person` but the type of the object that the variable `p` currently refers to is `Student`. Therefore, starting from the lowest class in the hierarchy of the objects, there is an `isOld` method in the `Student` class so that is the one that Java uses.

- The compiler does, however, look at the variable type and make sure the method exists in that type.
  e.g. Suppose the method
  ```
  public void foo()
  ```
  is only defined in the `Student` class. Then the code
  ```
  Person p = new Student();
  p.foo();
  ```
  would cause a compilation error since there is no `foo` method in the `Person` class. The compiler does not look at pervious lines of code so it is not sure what type `p` will have at execution time. The problem can be solved with a cast:
  ```
  Person p = new Student();
  ((Student)p).foo();
  ```

Overriding vs. Overloading

- **Overriding** occurs when two methods, one in the parent class, the other in the child class, have the same name and parameters. It allows a child class to provide a different implementation of a method that is already defined in the parent class.
- Overriding is a run-time concept. The object type, rather than the variable type, determines which overridden method is used in runtime. In the example above, the `isOld()` method in the `Student` class overrides the `isOld()` method in the `Person` class (since they have the same empty parameter list). Therefore `p.isOld()` calls the `isOld` method in the `Student` class based on the type of the object `p` holds at runtime (`Student`), as opposed to the type of the variable `p` (`Person`).
- **Overloading** occurs when two or more methods in the same class or different classes in the same class hierarchy have the same name but different parameters.
- Overloading is compile time concept. The variable type determines which overload method will be used at compile time.
  e.g. Suppose `equals` method has been written for `Person` and `Student` as follows:

```
class Person {
     // declaration of fields

     public boolean equals (Person other) {
          return other != null &&
                          name.equals(other.name);
     }
}

class Student extends Person{
     // declaration of additional fields

     public boolean equals (Student other) {
          return other != null &&
                     studentNum == other.studentNum;
     }
}
```

  In this case, the `equals` method in `Person` is overloaded because the `equals` method in `Student` has a different parameter (type). Therefore with the following declaration,

```
Person p = new Student();
Student s = new Student();
```

  `p.equals(s)` calls the `equals` method of `Person` because the type of variable `p` determines which overload method to run.

Using `instanceOf`

- Usage:
  ```
  variable instanceof Class
  ```
- `instanceof` evaluates to `true` or `false` depending on whether the variable refers to an object of type Class.
- `instanceof` also returns `true` if class of the object on the left is a child (or grandchild or greatgrandchild or …) of the class on the right.
- `instanceof` can be used to determine the type of object currently referred to by a reference variable.
  e.g.
  ```
  Person p;
  :
  :
  if (p instanceof Student) {
       System.out.println((Student)p).getNumber());
  }
  ```