Analyzing code with polymorphism, casting, and dynamic binding

Authors: Aleksandr Kalenchits, Brittney Bush

How to use this document

Below you will find a general outline of logical steps to help analyze code involving polymorphism, casting, and dynamic binding. The final section of this document provides examples of applying the outlined steps to problems.

When should you use this methodology?

You may recognize you are analyzing this type of code when a method is called on an object with different static and dynamic types.

Example:

Does the following code compile/run? What will be the output?

```
Teacher t = new TA();
t.learn();
```

The classes are defined as follows:

```
public abstract class Person {
   public abstract void move();
}

public class Teacher extends Person {
   public void move() {
      System.out.println("Teacher is moving");
  }

   public void teach() {
      System.out.println("Teacher is teaching");
   }
}

public class TA extends Teacher {
   public void move() {
      System.out.println("TA is moving");
   }

   public void learn() {
      System.out.println("I love to learn");
   }
}
```

The purpose of this guide is to explain which steps to take to arrive at the correct solution which is in this case "The code does not compile"

A few introductory tips

1. The code above **<u>does not compile and does not run</u>**. Hence, if you concluded that the code does not compile, there is no need for further thinking and explanation.
2. If there is at least **one compile error**, the code **does not compile**.
3. Code that compiles does not always run.
4. **<u>Casting will never change the dynamic type of an object.</u>**
5. If the code runs, do not forget to present the correct output if the question asks for it.
6. If the code errors during runtime, the **statements before the error will be executed** (printed[1], calculated, etc).
7. There may be deviations from the format of the code presented. For example, the declaration and instantiation of the variable can be split into 2 lines. Do not get confused, in this case, the procedure can still be applied to solve a problem.
8. **<u>Use the drawn hierarchy to help you!</u>**

What is the methodology?

In order to successfully analyze the code with polymorphism, casting and dynamic binding, we recommend using the following outline of logical steps. Once you get more comfortable with navigating these steps, you may omit trivial ones.

1. Draw the hierarchy of classes
    a. Include abstract **modifiers** as needed
    b. Include **methods** declared for each class
2. Determine the static and dynamic types of the reference to the object
    a. Static type is a type under which the variable is **declared (left side)** and is used at compile time
    b. Dynamic type is the type which was **instantiated (right side)** and is used at runtime
    ```
    [Static type] varName = new [Dynamic type]();
    ```
3. Determine whether the **code compiles**
    a. Examine the object declaration/instantiation line
        i. If no casting is present:
            1. The dynamic type must be the **same as the static type or a subclass further down the class hierarchy (i.e. a descendant class)**, otherwise, the code will not compile
            2. Attempting to instantiate an abstract class or interface throws a compile error
        ii. If explicit casting is present
            1. Please refer to Example 7-9 to see how casting during variable declaration will affect the analysis.
    b. Look at the method call
        i. If casting is present
            1. Check the relationship between the **static type** and the **cast type**. Up-cast always compiles, down-cast always compiles, side-cast never compiles
            2. Look at the method that is called on the cast object
                a. If the method is declared in the **cast class** or in some class **up from it in the hierarchy (i.e. an ancestor class)**, the method call compiles
                b. Otherwise, the method call does not compile

[1] Note, in some cases a print statement may be executed but the output may not be written to the console before the program terminates. This is due to the way the operating system buffers up work to do that's considered "expensive" (like console output).

ii. If no casting is present
1. Look at the method that is called on the object
a. If the method is declared in the **static type** class or in some class **up the hierarchy (i.e. an ancestor)**, the method call compiles

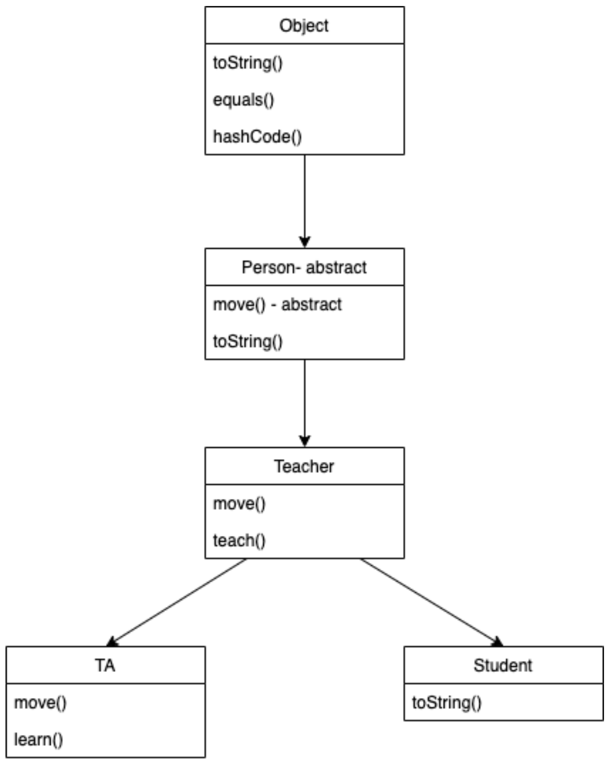
c. If no compiling issues were detected, the code compiles

4. **If the code compiles,** then trace the code to see if it runs
    a. The code will always run if there is no casting present
    b. **If casting is present**, look at the method call
        i. Look at the **cast type** and the **dynamic type**. If the dynamic type is the **same as cast type** or **down the hierarchy (i.e. a descendant)**, the code runs
        ii. Otherwise, the code generates a `ClassCastException` during runtime
5. Trace the output up to the end or up to the line with the runtime exception (refer to introductory tip 5)
    a. If the method is declared in the **dynamic type** class, it will be executed
    b. If the method is not declared in the **dynamic type** class, go up the hierarchy (i.e. look at ancestors one-by-one) until you find the method with the same signature.
    c. *If the code compiled based on step 3, then a method will always exist either in the dynamic class or in one of its ancestors*

Examples:

In the examples we will use the following class definitions:

```
public abstract class Person {
   public abstract void move();

   public String toString() {
      return "Person's toString()";
   }
}

public class Teacher extends Person {
   public void move() {
      System.out.println("Teacher is moving");
   }

   public void teach() {
      System.out.println("Teacher is teaching");
   }
}

public class TA extends Teacher {
   public void move() {
      System.out.println("TA is moving");
   }

   public void learn() {
      System.out.println("I love to learn");
   }
}
public class Student extends Teacher() {
   public String toString() {
      return "Student's toString()";
   }
}
```



<u>The first step of every example is omitted because the class hierarchy is drawn to the right of the class definitions. It may be helpful to screenshot or print this class diagram and have it close by when looking over the following examples</u>

Example 1:

```
Person p1 = new TA();
p1.learn();
```

Applying the steps:

2. Static type: **`Person`**
   Dynamic type: **`TA`**
3. Compilation
   a. First line: **`TA`** class is down the hierarchy from (i.e. a descendant of) the **`Person`** class, so the first line compiles
   b. Second line
      i. No casting: `learn()` is not declared in **`Person`** or any other classes up the hierarchy (i.e. ancestors) => **Compilation Error**

Since we have run into a compile error, the code does not compile and, therefore, does not run.

Answer: Does not compile, does not run.

Example 2:

```
Person p2 = new TA();
((Teacher) p2).teach();
```

Applying the steps:

2. Static type: **`Person`**
   Dynamic type: **`TA`**
3. Compilation
   a. First line: **`TA`** class is down the hierarchy from (i.e. a descendant of) the **`Person`** class, so the first line compiles
   b. Second line
      i. Casting present
         Teacher is down in the hierarchy from (i.e. a descendant of) **`Person`**, so the cast is a down-cast which always compiles
         `teach()` is defined in the Teacher class, so the method call compiles
   c. The code compiles
4. Running
   a. Casting present: dynamic type **`TA`** is down the hierarchy from (i.e. a descendant of) **`Teacher`**, so casting runs
   b. Code runs
5. Output
   a. `teach()` is not declared in the **`TA`** class
   b. `teach()` is declared closest in the hierarchy in the **`Teacher`** class, so Teacher's `teach()` will execute
   c. The code will print "Teacher is teaching" to the console

Answer: The code compiles and runs. It prints "Teacher is teaching"

Example 3:

```
Teacher p3 = new TA();
p3.teach();
```

Applying the steps:

2. Static type: **`Teacher`**
   Dynamic type: **`TA`**
3. Compilation
   a. First line: **`TA`** class is down the hierarchy from (i.e. descendant of) the **`Teacher`** class, so the first line compiles
   b. Second line
      i. No casting: `teach()` is declared the **`Teacher`** class, so the code compiles
   c. Both lines compile, so the code compiles
4. Running
   a. No casting is present, so the code does not throw a `RuntimeException`
5. Output
   a. `teach()` is not declared in the **`TA`** class
   b. `teach()` is declared closest in the hierarchy in the **`Teacher`** class, so Teacher's `teach()` will execute
   c. The code will print "Teacher is teaching" to the console

Answer: The code compiles and runs. It prints "Teacher is teaching"

Example 4:

```
Student p4 = new TA();
p4.move();
```

Applying the steps:

2. Static type: **`Student`**
   Dynamic type: **`TA`**
3. Compilation
   a. First line: **`TA`** class is **not down the hierarchy** from **(i.e. not a descendant of)** the **`Student`** class, so the first line does not compile
   b. Compilation Error

Answer: The code does not compile and does not run

Example 5:

```
Teacher p5 = new Student();
p5.toString();
```

Applying the steps:

2. Static type: **`Teacher`**
   Dynamic type: **`Student`**

3. Compilation
    a. First line: **`Student`** class is down the hierarchy from (i.e. a descendant of) the **`Teacher`** class, so the first line compiles
    b. Second line
        i. No casting: `toString()` is not declared the **`Teacher`** class, but it is declared in the **`Person`** class which is up the hierarchy from (an ancestor of) **`Teacher`**, so the line compiles.
    c. Both lines compile, so the code compiles
4. Running
    a. No casting is present, so the code does not throw a `RuntimeException`
5. Output
    a. `toString()` is declared in the Student class
    b. The code will return a **`String`** “Student’s toString”

Answer: The code compiles and runs. It returns the String “Student’s toString”

Example 6:

```
Person p6 = new Teacher();
((TA) p6).move();
```

Applying the steps:

2. Static type: **`Person`**
   Dynamic type: **`Teacher`**
3. Compilation
    a. First line: **`Teacher`** class is down the hierarchy from (i.e. the descendant of) the **`Person`** class, so the first line compiles
    b. Second line
        i. Casting present
           `TA` is down in the hierarchy from (i.e. the descendant of) **`Person`**, so the cast is a down-cast which always compiles
           `move()` is defined in the **`TA`** class, so the method call compiles
    c. The code compiles
4. Running
    a. Casting present: dynamic type **`Teacher`** is above the hierarchy from (i.e. an ancestor of) **`TA`**, so **casting does not run**
    b. Code throws a **`ClassCastException`**

Answer: The code compiles and does not run. It generates a `ClassCastException` during runtime

Example 7*:

```
Person p7 = (Teacher) new TA();
((Person) p7).move();
```

This is a special type of the code analysis

We can rewrite the code in the following way:

```
TA intermediateObject = new TA();
Person p7 = (Teacher) intermediateObject;
((Person) p7).move();
```

Important note: technically, the codes are not equivalent because the original prompt has only one object declared in memory while rewritten version has two objects declared in memory. However, for the purpose of compiling/running analysis, the codes are identical
Given the rewritten code, we should **apply the methodology to both declarations**

Applying the steps:

2. Static type `intermediateObject`: **`TA`**
   Dynamic type `intermediateObject`: **`TA`**

   Static type `p7`: **`Person`**
   Dynamic type `p7`: **`TA`** (refer to introductory tip 4)
3. Compilation
   a. Declarations
      i. The first declaration has the same static/dynamic types, so it compiles
      ii. Let's move to casting. The object being cast is `intermediateObject.` Its static type is **`TA`** and the cast type is **`Teacher`**, which is an upcast, so the line compiles
      iii. Now, we are assigning an object cast to **`Teacher`** to a statically **`Person`** variable. This step resembles step 3a in the original methodology**. For compilation assessment only** we can pretend **`Teacher`** is our dynamic type and **`Person`** is a static type. **`Teacher`** is down the hierarchy from (i.e. descendant of) **`Person`**, so the assignment is valid.
   b. Method call line
      i. Casting present
         Cast type **`Person`** is the same as the static type of `p7`, so the cast compiles
         `move()` is defined in the **`Person`** class, so the method call compiles
   c. Both lines compile, so the code compiles
4. Running
   a. First cast examination:
      i. Dynamic type of `intermediateObject` is **`TA`**, cast type is **`Teacher`**, dynamic type is down the hierarchy from (i.e. a descendant of) the cast type, so this cast runs
   b. Second cast examination:
      i. Dynamic type of `p7` is **`TA`**, cast type is **`Person`**, dynamic type is down the hierarchy from (i.e. descendant of) the cast type, so this cast also runs
5. Output
   a. `move()` is declared in the **`TA`** class
   b. The code will print "TA is moving" to the console

Answer: The code compiles and runs. It prints "TA is moving"

Example 8*:

```
Teacher p8 = (Person) new TA();
((TA) p8).learn();
```

This is a special type of the code analysis

We can rewrite the code in the following way:

```
TA intermediateObject = new TA();
Teacher p8 = (Person) intermediateObject;
((TA) p8).learn();
```

Important note: technically, the codes are not equivalent because the original prompt has only one object declared in memory while rewritten version has two objects declared in memory. However, for the purpose of compiling/running analysis, the codes are identical

Given the rewritten code, we should **apply the methodology to both declarations**

Applying the steps:

2. Static type `intermediateObject`: **`TA`**
   Dynamic type `intermediateObject`: **`TA`**

   Static type `p8`: **`Teacher`**
   Dynamic type `p8`: **`TA`** (refer to introductory tip 4)
3. Compilation
   a. Declarations
      i. The first declaration has similar static/dynamic types, so it compiles
      ii. Let's move to casting. The object being cast is `intermediateObject`. Its static type is **`TA`** and the cast type is **`Person`** which is an upcast, so the line compiles fine
      iii. Now, we are assigning an object cast to **`Person`** to a statically **`Teacher`** variable. This step resembles step 3a in the original methodology**. For compilation assessment only** we can pretend **`Person`** is our dynamic type and **`Teacher`** is a static type. **`Person`** is not down the hierarchy from (i.e. not a descendant of) **`Teacher`**, so the assignment does not compile
   b. Compilation error

Answer: The does not compile and does not run.

Example 9*:

```
Object p9 = (TA) new Teacher();
System.out.println(p9);
```

This is a special type of the code analysis

We can rewrite the code in the following way:

```
Teacher intermediateObject = new Teacher();
Object p9 = (TA) intermediateObject;
System.out.println(p9);
```

Important note: technically, the codes are not equivalent because the original prompt has only one object declared in memory while rewritten version has two objects declared in memory. However, for the purpose of compiling/running analysis, the codes are identical

Given the rewritten code, we should **apply the methodology to both declarations**

Applying the steps:

2. Static type `intermediateObject`: **`Teacher`**
   Dynamic type `intermediateObject`: **`Teacher`**

   Static type `p9`: **`Object`**
   Dynamic type `p9`: **`Teacher`** (refer to introductory tip 4)

3. Compilation
    a. Declarations
        i. The first declaration has similar static/dynamic types, so it compiles
        ii. Let's move to casting. The object **`Teacher`** and the cast type is **`TA`**, which is a downcast, so it compiles
        iii. Now, we are assigning an object cast to **`TA`** to a statically **`Object`** variable. This step resembles step 3a in the original methodology**. For compilation assessment only** we can pretend **`TA`** is our dynamic type and **`Object`** is a static type. **`TA`** is down the hierarchy from (i.e. a descendant of) **`Object`**, so the assignment is valid.
    b. Method call line
        i. No casting: `toString()` is declared in the Object class, so the call compiles
4. Running
    a. First cast examination:
        i. The dynamic type of `intermediateObject` is **`Teacher`** and the cast type is **`TA`**. The dynamic type is up the hierarchy from (i.e. an ancestor of) the cast type, so there will be a runtime error
        ii. **`ClassCastException`**

Answer: The code compiles and throws a `ClassCastException` during runtime