# CS1101 Programming and Problem Solving

Dr. Gina Bai

Spring 2023

## Logistics

- **ZY-9** on zyBook > Assignments
  - Due: **Wednesday, April 19**, at 11:59pm
- **PA11 – A, B** on zyBook > Chap 11
  - Due: **Thursday, April 20**, at 11:59pm
- **ZY-10** on zyBook > Assignments
  - Due: **Monday, April 24**, at 11:59pm

**CS1101 Recitation Eval**
**Anonymous**
Attended or not attended
Your feedback is greatly appreciated!

# Four Main Principles of OOP

1. **Abstraction**

   To simplify reality and focus only on the properties and external behaviors rather than inner details

2. **Encapsulation**

   Hiding the implementation details (data and the programs that manipulate the data) of an object from the clients of the object

3. **Inheritance**

   A class can derive its methods and properties from another class

4. **Polymorphism**

# Polymorphism

zyBook Chap 10.5

# Polymorphism

- poly-morphism → **many forms**
- A class can implement an **inherited** method in **its own way**
  - That is, **override** the inherited method. Such as toString() and equals()
- Allows a **variable** of a **superclass type** to **refer to** an object of **one of its subclasses**
  - `<SuperclassType> <objName> = new <SubclassName>();`

## Polymorphism – Reference Types

Polymorphism refers to **determining which overridden method to execute depending on data types**

- `<SuperclassType> <objName> = new <SubclassName>();`

```
Person p1 = new Person(); // Declared as a Person, refers to a Person
System.out.println(p1.toString()); // the Person version toString() will be called

Person p2 = new Student(); // Declared as a Person, refers to a Student
System.out.println(p2.toString()); // the Student version toString() will be called
```

```
You-Know-Who (age: 0)
You-Know-Who (age: 0) is a Student
```

# Why Polymorphism

- The container class (a special component that can hold the gathering of the components – not covered in CS1101) can use the parent type in the composition relationship (has-a relationship)

```java
// This Person array allows reference to all these
// different types of objects that are instances of its subclasses
Person[] newToVandy = new Person[3];

newToVandy[0] = new Student(18, "new Stu", "nS");
newToVandy[1] = new Employee(28, "new Emp", "nE");
newToVandy[2] = new Faculty(38, "new Fac", "nF");
System.out.println(Arrays.toString(newToVandy));
```

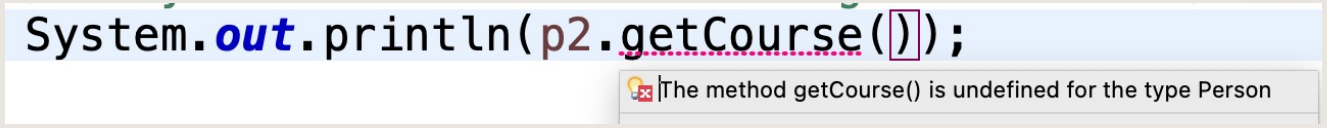
```
[new Stu (age: 18) is a Student, new Emp (age: 28) is an Employee, new Fac (age: 38) is an Employee – Faculty]
```

# Polymorphism – Reference Types

```
Person p1 = new Person(); // Declared as a Person, refers to a Person
System.out.println(p1.toString()); // the Person version toString() will be called

Person p2 = new Student(); // Declared as a Person, refers to a Student
System.out.println(p2.toString()); // the Student version toString() will be called

System.out.println(p2.getCourse());
```



```
System.out.println(p2.getCourse());
```

The method getCourse() is undefined for the type Person

Though a Person object can **refer** to a Student object, and **perform** the **overridden version** of certain methods in Student Class (e.g., toString()), a Person object **CANNOT perform the methods ONLY** in the Student Class

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Does Tulip have a toString()?
No

Does Tulip explicitly have a toString()?
No
Look into its superclass

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("=====================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
Tulip 1
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("=====================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
Tulip 1
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
Tulip 1
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("=====================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
Tulip 1
Lily 2
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Which method1 will be printed?
Lily's version or Tulip's version?

```
pretty[0]: Rose Lily
Tulip 1
Lily 2
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Since the pretty[0] refers to a Tulip, and Tulip Class contains an overridden method1()

```
pretty[0]: Rose Lily
Tulip 1
Lily 2
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

Since the pretty[0] refers to a Tulip, and Tulip Class contains an overridden method1()

```
pretty[0]: Rose Lily
Tulip 1
Lily 2
Tulip 1
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
Tulip 1
Lily 2
Tulip 1
======================
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

```java
public class Violet {

    public void method1() {
        System.out.println("Violet 1 ");
    }

    public void method2() {
        System.out.println("Violet 2 ");
    }

    public String toString() {
        return "Violet";
    }
}

public class Rose extends Lily {

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

```java
public class Lily extends Violet {

    public void method1() {
        super.method1();
        System.out.println("Lily 1 ");
    }

    public void method2() {
        System.out.println("Lily 2 ");
        method1();
    }

    public String toString() {
        return "Lily";
    }
}

public class Tulip extends Rose {

    public void method1() {
        System.out.println("Tulip 1 ");
    }
}
```

```
pretty[0]: Rose Lily
Tulip 1
Lily 2
Tulip 1
======================
pretty[1]: Lily
Violet 1
Lily 1
Lily 2
Violet 1
Lily 1
======================
pretty[2]: Violet
Violet 1
Violet 2
======================
pretty[3]: Rose Lily
Violet 1
Lily 1
Lily 2
Violet 1
Lily 1
======================
```

Q: What's the output of the client code? →

```java
Violet[] pretty = {new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; ++i) {
    System.out.println("pretty[" + i + "]: " + pretty[i]);
    pretty[i].method1();
    pretty[i].method2();
    System.out.println("======================");
}
```

# Summary – Four Main Principles of OOP

1. **Abstraction**

   To simplify reality and focus only on the properties and external behaviors rather than inner details

2. **Encapsulation**

   Hiding the implementation details (data and the programs that manipulate the data) of an object from the clients of the object

3. **Inheritance**

   A class can derive its methods and properties from another class

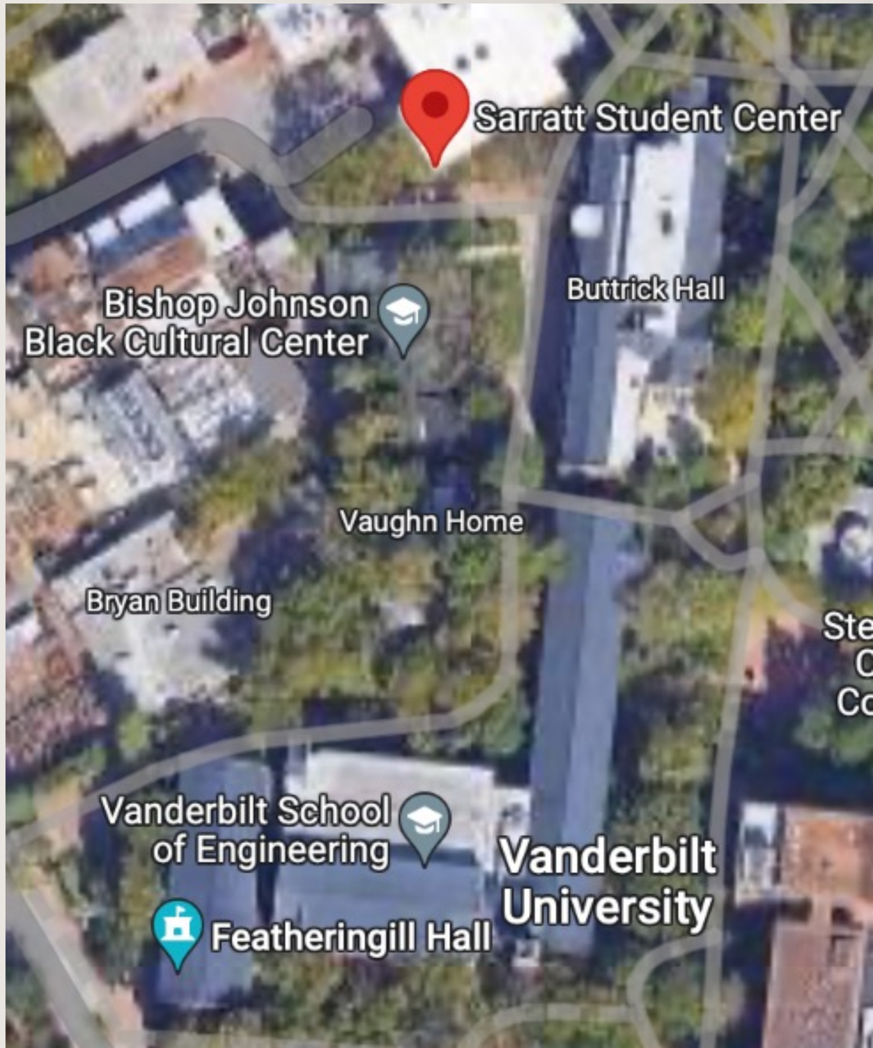
4. **Polymorphism**

   Overriding methods among subclasses, and determining which overridden method to execute depending on data types

## About the Final Exam…

## Format

- **Paper-based**, closed book, closed notes
- Short answer, multiple choice, true/false, code reading and writing **(a lot)**
- **Friday, April 28, 7pm – 10pm**
  - **Sarratt Cinema**
  - Arrive early!!!
  - No alternate dates/times provided
- Includes a *reference guide*



## Learning Objectives – Chap 1

- General computer science terminology
- Java terminology (e.g., compile, bytecode, JVM, interpreter…)
- Java program structure (e.g., class, method, variable, header…)
- Using a `Scanner` object to get input from the user
  - Prompt for user input first, and then read in input with Scanner
- Output statements using `print`, `println`, and `printf`
- Types of programming errors: syntax, runtime and logic
  - Identify, and describe if the program would compile and run

## Learning Objectives – Chap 2

- Java identifiers (e.g., naming convention)
- Declaring, initializing/assigning, and using variables
- Various data types (primitive data types and objects)
- Expressions
- Arithmetic operators and operator precedence
- Type conversion: promotion/coercion and type casting
- `Math` class methods
- Escape sequence

## Learning Objectives – Chap 3

- Problem decomposition (e.g., create classes and methods)
- Variable scope
- Declaring methods and calling methods
    - Return type, parameters, data type of the parameters
- Using parameters to pass information to a method
    - Primitive data types – pass by value
    - Objects – pass by reference (its value in the stack is the reference)
- Returning a value from a method

# Learning Objectives – Chap 4

- Decision statement structure - `if` statements
- Equality, Relational and Logic operators, and operator precedence
- Short circuit evaluation
- The `boolean` type
- String methods
  - length, equals, equalsIgnoreCase, toLowerCase, toUpperCase, trim, etc…
  - split (split a String into a String array)
- The `char` type and the `Character` wrapper class methods
  - isDigit, isLetter, toLowerCase, toUpperCase, etc…

# Learning Objectives – Chap 5

- `while` loops, `do-while` loops, `for` loops
  - Nested loops
  - Fencepost problems, and sentinel values
- `Scanner` next methods
  - nextInt, nextDouble, next, nextLine()
  - next() vs. nextLine() → also commonly used to discard invalid input
- The `Random` object and generating random numbers
- Assertions

# Learning Objectives – Chap 6

- File input
  - FileInputStream
  - `Scanner` next and hasNext methods
  - Token-based processing
  - Line-based processing
    - Creating a `Scanner` object on a `String` and tokenize it
- File output
  - FileOutputStream
  - PrintWriter

# Learning Objectives – Chap 7

- Array construction
  - Shorthand vs. for loop
  - Arrays of primitive data type
  - Arrays of objects
    - Two-phase initialization: construct the array first, construct each element
    - Use if statement to check if an element is null or not before accessing it
- Array bounds and array indices
  - The `length` field
  - Accessing and processing array elements
- Passing entire arrays to a method
- Returning entire arrays from a method

# Learning Objectives – Chap 7

- Array modification
  - Search an element in an array
    - Sequential
    - Binary search
  - Swap array elements
  - Sort an array
    - Selection sort
    - Insertion sort
- The `Arrays` class and its methods (e.g., `fill`, `sort`, `equals`, `toString`, etc…)

## Learning Objectives – Chap 9

- Class
  - Instance variables and methods
  - Static fields and methods
  - Constructors
    - Default constructor
    - Overloading constructors
  - Implicit parameter `this`
- Access modifiers → public vs. private vs. protected
  - Encapsulation (information hiding)
  - Accessors and mutators

# Learning Objectives – Chap 10

- Why inheritance is useful
- Class hierarchies → Base class (superclass) vs. derived class (subclass)
  - Subclass **extends** superclass
  - ***Is-a*** vs. ***has-a*** relationships
  - Use of keyword super
- Polymorphism
  - Declared variable type determines which methods are available
  - Actual type of object determines **which overridden method** gets invoked
- Overriding methods (particularly toString and equals)
- Overriding vs. overloading

# Learning Objectives – Chaps 9 & 10

- Be able to 1) explain, 2) discuss the benefits, and 3) implement code that demonstrate the principles of Object-Oriented Programming
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

## Exam Prep

Be familiar with the following:

- All programming assignments you have completed
- All demonstrations done in class
- All problems in Midterm Exams 1 & 2
- Understand what was done and why
- Understand how they work

## Code Reading

- Read segments of Java code and
    - Determine the output they produce
    - Determine the value of variables
    - Determine the value of expressions
    - Find program errors
    - Trace through control flow constructs

## Code Writing

- Write segments of Java code
    - Write simple expressions
    - Declare and initialize variables
    - Write control flow constructs to do conditionals and/or loops
    - Produce the requested output
    - Write a class definition with appropriate instance variables and methods
        - Public vs. private
        - Accessor and mutators
        - Inheritance, derived classes