2023-04-17

# Upcoming Schedule

| Monday | Wednesday | Friday |
|---|---|---|
| **04/17/2023 (90)**<br>• Review: Unit 5, Unit 9<br>**• AP CS Question 2: Classes** | **04/19/2023 (90)**<br>• Review: Units 6-7<br>**• AP CS Question 3: Array/ArrayList** | **04/21/2023 (45)**<br>• Review: Unit 10<br>• More recursion exercises like we did on Apr-7 |
| **04/24/2023 (90)**<br>• Review: Unit 8<br>**• AP CS Question 4: 2D Array** | **04/26/2023 (90)**<br>**• AP CS Multiple Choice Game** | **04/28/2023 (45)**<br>• Review: Unit 7, Unit 10<br>• Algorithms: Iterative/recursive binary search, selection sort, insertion sort, merge sort |
| **05/01/2023**<br>**• FINAL** | **05/03/2023**<br>**• AP EXAM** | |

# Unit 5 and Unit 9
# AP CS FRQ 2
(Classes and Inheritance)

# Object-Oriented Programming (OOP)

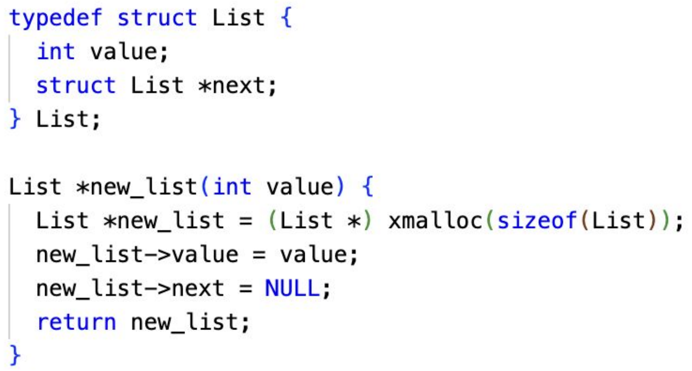
OOP is a programming methodology. It introduced the concepts of classes and objects.

Before OOP, the dominant paradigm was "structured programming." Code and data were separate. You'd organize your code into procedures/functions. For data, many languages had "records" or "structs"... essentially objects with just variables and no methods. OOP had the idea of packaging the methods and variables together.

OOP encourages you to think about objects as "actors" that you command to take actions, but how they do it internally is abstracted away.

OOP, like grunge, became huge in the early 90's, with languages like C++ and Smalltalk, and yes, Java.

It's still popular today, because it proved to be a great way to organize and maintain large, complex programs.



```c
typedef struct List {
  int value;
  struct List *next;
} List;

List *new_list(int value) {
  List *new_list = (List *) xmalloc(sizeof(List));
  new_list->value = value;
  new_list->next = NULL;
  return new_list;
}
```

*Old School Structured Programming: Creating a linked list in C (not C++)*

## The Four Pillars of Object-Oriented Programming (OOP)

| | |
|---|---|
| **Abstraction** | Objects provide an ***interface*** that external code interacts with.<br>The ***implementation*** of the object is not of concern to external code; it is abstracted away, and could be changed without any alteration needed to calling code. |
| **Encapsulation** | The internal state of an object, such as variables, are bundled with it and hidden from external code. |
| **Inheritance** | Objects belong to classes, and classes inherit from other classes.<br>Objects inherit the methods and variables of their superclasses.<br>*Related concept:* Containment/Composition is when objects contain other objects. |
| **Polymorphism** | `Circle` and `Square` inherit from `Shape`, which defines a `draw` method.<br>Each subclass of `Shape` implements the `draw` method with the code to draw the right shape.<br>Calling code, working with collections of `Shape`, is indifferent to the particular type of `Shape` being drawn. |

# Definitions

```java
public class Dog {
  private String name;
  private int age;
  private String breed;

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

  public void eat() { /* ... */ }
  public void play() { /* ... */ }
  public void sleep() { /* ... */ }

  @Override
  public String toString() {
    return "A " + age + "-year old " + breed + " named " + name;
  }
}
```

**Class**: A blueprint for a type of object.

**Object instance,** or just **object**: An instantiation (copy) of a class created with the `new` operator. There can be many of them:

```java
Dog roscoe = new Dog("Roscoe", 3, "German Shepherd");
Dog lucy = new Dog("Lucy", 1, "Golden Retriever");
```

**Instance variables**, also known as **fields, attributes,** or **properties:** Every instance gets their own copy of these variables. They hold the data for each object instance.

**Constructor:** Code that is run to initialize the object, optionally taking parameters passed using `new`.

**Instance Methods:** Operations that the object can perform. When instance variables are used, they are fetched from the instance the method was invoked on.

## The Anatomy of a Constructor

Things to keep in mind:

The constructor name must **always** match the name of the class.

Always prepend the **public** keyword before your constructor name. (`private` constructors have their uses, but are less common.)

Constructors have **no return type**. **Not even void!**

The constructor definition is often included before any other method definitions (but is not required to be)

```java
public class ClassName {

    // Instance Variable Declarations

    // Constructor - same name as Class, no return type
    public ClassName() {
      // Implementation not shown
    }

    // Other methods ...
}
```

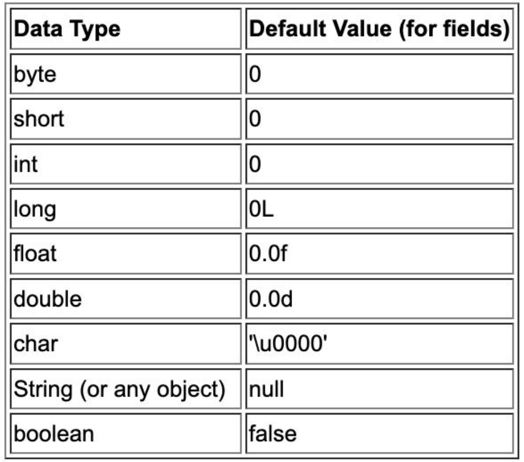
# Instance Variable Initialization

Instance variables are typically assigned initial values by a constructor.

They can also be initialized as part of their declaration:

```
private boolean isAlive = true;
```

If you do not initialize an instance variable, it will be initialized to the default value for its type.



| Data Type | Default Value (for fields) |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |

```
public class Person {
  private String name;
  private int age;
  private boolean isAlive = true;

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

## Types of Constructors

- **Default Constructor**
- No-Argument Constructor
- Parameterized Constructor
- Overloaded Constructors

```
public class Person
{
  // No constructor defined
  // Java creates a default constructor
}
```

- Automatically generated by the Java compiler when you do not supply a constructor
- Does not do any kind of specialized initialization of the instance beyond whatever in-place variable initialization exists, and default values

## Types of Constructors

- Default Constructor
- **No-Argument Constructor**
- Parameterized Constructor
- Overloaded Constructors

```
public class Person {
  private String name;
  private int age;

  public Person() {
    name = "Billy";
    age = 25;
  }
}
```

- A constructor you define that takes no arguments
- Can perform any kind of initialization that the the instance requires
- No parameters can be passed in during **`new`** to customize the instance

***When you define a Constructor (any kind)***
***Java will NOT create a Default Constructor!***

## Types of Constructors

- Default Constructor
- No-Argument Constructor
- **Parameterized Constructor**
- Overloaded Constructors

```
public class Person {
  private String name;

  public Person(String initName) {
    name = initName;
  }
}
```

- A constructor you define that takes arguments
- Can perform any kind of initialization that the the instance requires
- Each instance can be initialized individually based on the values passed into **`new`**

## Types of Constructors

- Default Constructor
- No-Argument Constructor
- Parameterized Constructor
- **Overloaded Constructors**

```
public class Person {
  private String name;
  private int age;

  public Person() {
    name = "Name unknown";
  }
  public Person(String initName) {
    name = initName;
    age = 30;
  }
  public Person(String initName, int initAge) {
    name = initName;
    age = initAge;
  }
}
```

- A class may have multiple constructors
- Each constructor **must**
  - have the same name as the class;
  - declare no return type
  - have a distinct set of parameter types
- This is quite common. Sometimes, you want to be able to create an object with a default set of values, but also have the ability to specify values.

# Types of Constructors

- Default Constructor
- No-Argument Constructor
- Parameterized Constructor
- **Overloaded Constructors**

```
public class Person {
  private String name;
  private int age;

  public Person() {
    this("Name unknown");
  }
  public Person(String initName) {
    this(initName, 30);
  }
  public Person(String initName, int initAge) {
    name = initName;
    age = initAge;
  }
}
```

One last cool trick...

You can call `this()` as a method within constructors, and it will chain to another constructor.

This is like `super()` constructor chaining, but it chains to a constructor in the same class, not a superclass. Like `super()`, it has to be the first statement in the constructor body.

Not needed for the AP exam.

I don't see people do it in the wild much, either... maybe they don't know about it :)

# Preconditions and Postconditions

DEFINITELY on AP exam

## Preconditions and Postconditions

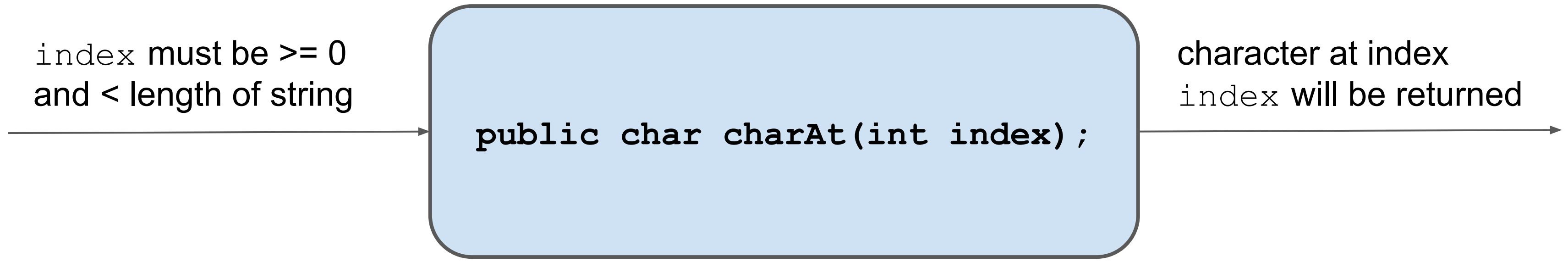
**Preconditions** and **postconditions** are a "contract" that describes what a method requires about its inputs, and what it promises as output.

**Precondition**

`index` must be >= 0 and < length of string

**Postcondition**

character at index `index` will be returned



```
public char charAt(int index);
```

*The caller must satisfy this requirement when calling* `charAt.`

*In return, the desired character will be returned.*

## Preconditions

Preconditions are part of the method's documentation, and may exist only as comments.

***There is no expectation that the method will check to ensure preconditions are satisfied.***

They may or may not be enforced by the method's code – the programmer using the method should read the documentation and understand the "contract" the method offers.

```
/**
 * Precondition: num2 is not zero.
 * Postcondition: Returns the quotient of num1 and num2.
 */
public double divide(double num1, double num2)
{
   return num1 / num2;
}
```

## Who enforces preconditions?

Sometimes preconditions ARE enforced by the method's code.

An actual implementation of Java's `String.charAt`:

```
public char charAt(int index) {
      if ((index <  0) || (index >= value.length)) {
            throw new StringIndexOutOfBoundsException(index);
      }
      return value[index];
}
```

Here, an exception is thrown if the precondition is not satisfied.

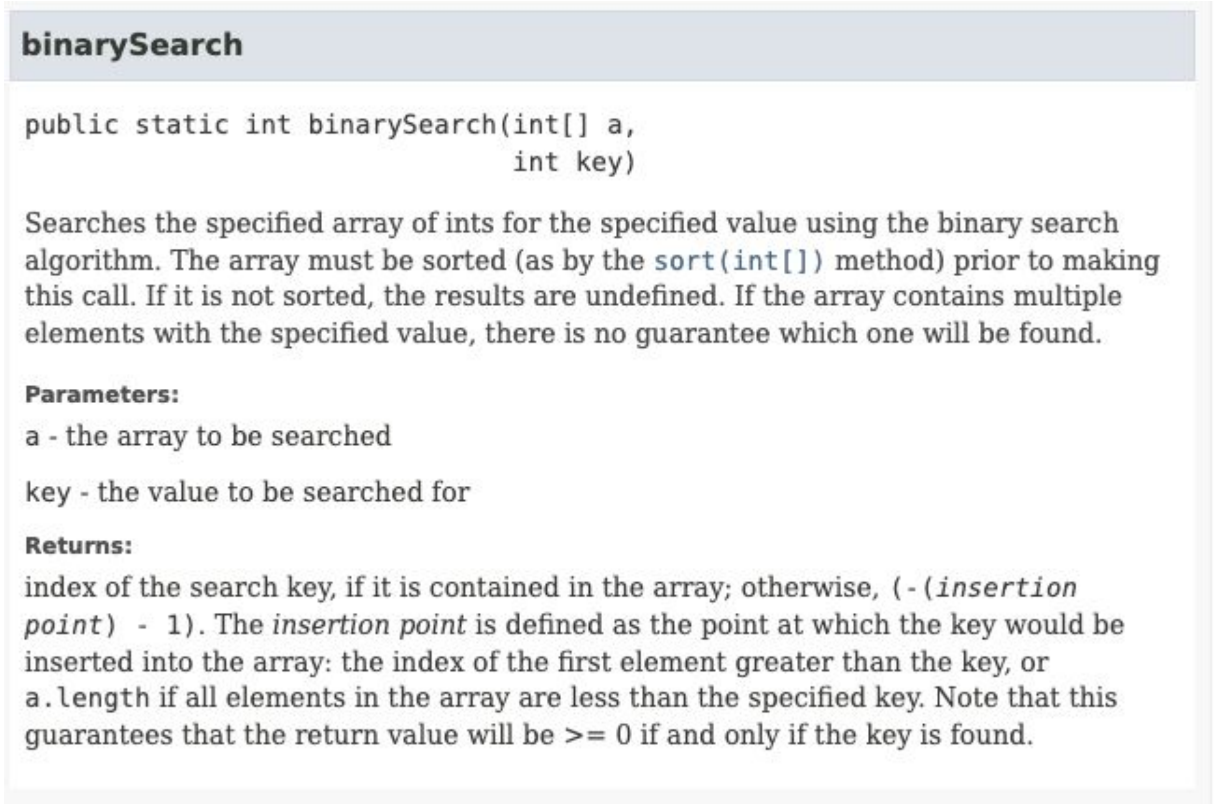
Throwing an exception in Java is a common way to handle failed preconditions.

## Who enforces preconditions?

At other times, it is not reasonable for the method to enforce the precondition.

The programmer calling the method must understand the preconditions and satisfy them.

The precondition here exists only as documentation, and describes the consequences of failing to meet it. (Where is the precondition?)



**binarySearch**

```
public static int binarySearch(int[] a,
                               int key)
```

Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the `sort(int[])` method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

**Parameters:**

a - the array to be searched

key - the value to be searched for

**Returns:**

index of the search key, if it is contained in the array; otherwise, `(-(`*`insertion point`*`) - 1)`. The *insertion point* is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or `a.length` if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

## Postconditions

A **postcondition** is a condition that is true after running the method. It is what the method promises to do.

Postconditions describe the outcome of running the method, for example what is being returned or the changes to the instance variables.

Examples:

- **`String.compareTo()`** The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters).
- **`Math.random`** Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

## Access Control: `private` and `public`

**`private`**

A `private` method or variable may only be accessed by code within the class.

- On the AP Exam, and most of the time IRL, all instance variables should be declared `private`.
- Even a subclass of your class cannot access `private` variables!
- If a method isn't part of the **interface** that other code should call, that is, if it's part of its internal **implementation**, it probably should be `private`.

**`public`**

A `public` method or variable may be accessed by any code, inside or outside of the class.

## Accessor Methods

It is best practice to define your instance variables as `private`.

Still, code outside your class often needs to get the values in those variables.

For this reason, classes often define ***accessor methods***, which are `public` methods that provide read-only access to `private` instance variables.

Sometimes, these are called "get methods" or "getters."

The return type matches the type of the variable being returned.

```
public class Person {
  private String name;

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

  public String getName() {
    return name;
  }
}
```

# Accessor Methods

Accessor methods can also return a filtered or transformed version of the `private` variable.

This is the power of accessor methods – they abstract away the variable itself, so the way the phone number is represented here could be changed.

```
public class Person
{
  private String phoneNumber;
   ...
  public String getAreaCode() {
    return phoneNumber.substring(0,3);
  }
}
```

## Mutator Methods

Mutator methods are the opposite of accessor methods...

They are `public` methods that modify internal `private` instance variables.

Sometimes these are called "set methods" or "setters."

These `public` methods typically have `void` return type, and take one parameter with the same type as the `private` instance variable being modified.

```
public class Person {
  private String name;

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

  public String getName() {
    return name;
  }

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

## Mutator Methods

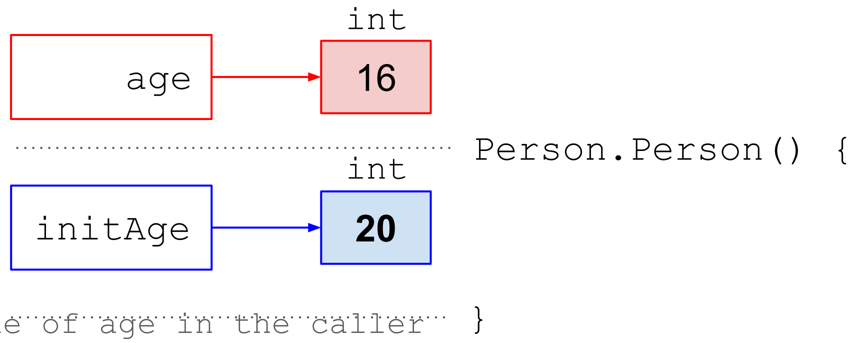
Mutator methods can also filter, verify, or transform a value before assigning it to a private instance variable value.

```
public class Person
{
  private String areaCode;
   ...
  public void setAreaCode(String phoneNumber) {
    areaCode = phoneNumber.substring(0,3);
  }
}
```

## Method Parameters: **Pass by Value** / Pass by Reference

- Parameters of a method behave like local variables of the method. Each actual parameter is copied into one of these variables.
- For primitive types, the variable contains a copy of the actual parameter. The method may change it, but it has no effect on any variables in the calling code.

```
int age = 16;
Person p = new Person(age);
// age is guaranteed to still be 16

public class Person {
  public Person(int initAge) {
    // initAge is a COPY of age
    initAge = 20; // does not alter the value of age in the caller
  }
}
```

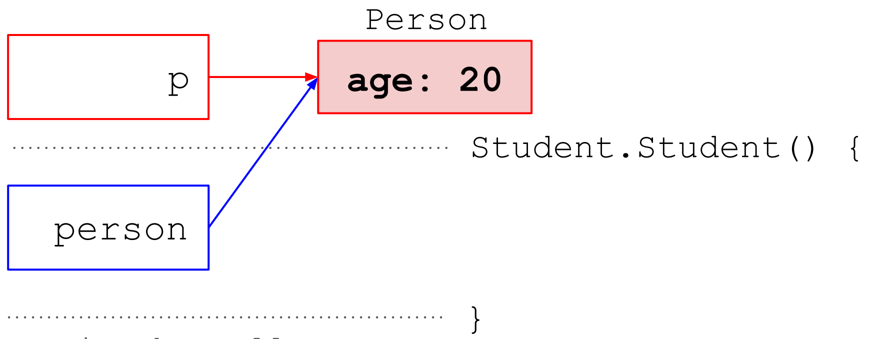


## Method Parameters: Pass by Value / **Pass by Reference**

For object types, what is passed into the method is a reference to an object.

If the referenced object is mutable in any way, the method could make changes to it, and the caller would see those changes after the method returns.

```
int age = 16;
Person p = new Person(age);
// p.age == 16
Student s = new Student(p);
// p.age == 20

public class Student {
  public Student(Person person) {
    // person is a REFERENCE to the same object in the caller
    person.age = 20; // alters the Person passed in from the the caller
  }
}
```



## `static`

`static` variables and methods are at class-level, not instance-level

- There is only one copy of a `static` variable
- `static` methods aren't run in the context of any particular instance (no `this`)
- They can be `public` or `private`
- Often, `static` variables are `CONSTANT_VALUES` and declared `final`

When using the dot operator, you typically use the name of the class, instead of a particular instance.
(You can legally say *`instance`*`.some_static`, but it could just be confusing.)

- `Math.PI`
- `Math.random()`
- `Math.sqrt()`

Within a class, you can just refer to a static member by name.

There's also the `import static` shorthand: `import static java.lang.Math.*;`

The `main` method is a `static` method. The JVM invokes it without `new`-ing up your main class.

# Statics

- **Statics can directly access other Statics**
- Statics cannot directly access non-Statics
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person.getNumPeople();

A> 0
```

## Statics

- Statics can directly access other Statics
- **Statics cannot directly access non-Statics**
- Non-Statics can directly access Statics

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
public static int getNumPeople() {
  // System.out.println(name); <- Nope.
  return numPeople;
}
```

# Statics

- Statics can directly access other Statics
- Statics cannot directly access non-Statics
- **Non-Statics can directly access Statics**

```
class Person {
  private static int numPeople = 0;
  private String name;
  public Person(String initName) {
    numPeople++;
    name = initName;
  }
  public static int getNumPeople() {
    // System.out.println(name);
    return numPeople;
  }
  public void report() {
    System.out.println(name + " is one of " + numPeople + " people");
  }
}
```

```
Person p1 = new Person("Julie");
Person p2 = new Person("Bobby");

p1.report()

B> Julie is one of 2 people

p1.getNumPeople();

C> 2
```

## Scope – what methods and variables are accessible

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private String email;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

## Scope – what methods and variables are accessible

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```java
public class Person {
  private String name;
  private String email;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

## Scope – what methods and variables are accessible

- **Class Level Scope** Instance and static variables inside a Class.
- **Method Level Scope** Local variables (including parameter variables) inside a method.
- **Block Level Scope** Loop variables and other local variables defined inside of blocks of code with { }

```
public class Person {
  private String name;
  private String email;

  public void print(int length) {
    for (int i = 0; i < length; i++) {
      System.out.println(name.charAt(i));
    }
  }
}
```

## OOP & Inheritance

**Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.

## OOP & Inheritance

**Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.

| Person |
|---|
| *name* |
| *address* |

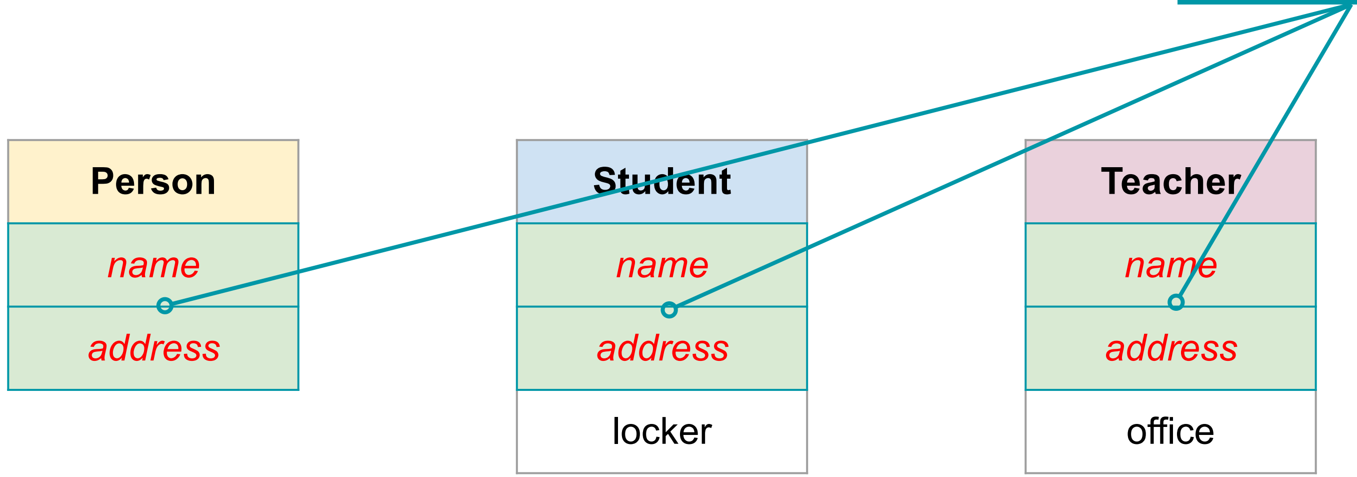
| Student |
|---|
| *name* |
| *address* |
| locker |

| Teacher |
|---|
| *name* |
| *address* |
| office |

# OOP & Inheritance & Generalization

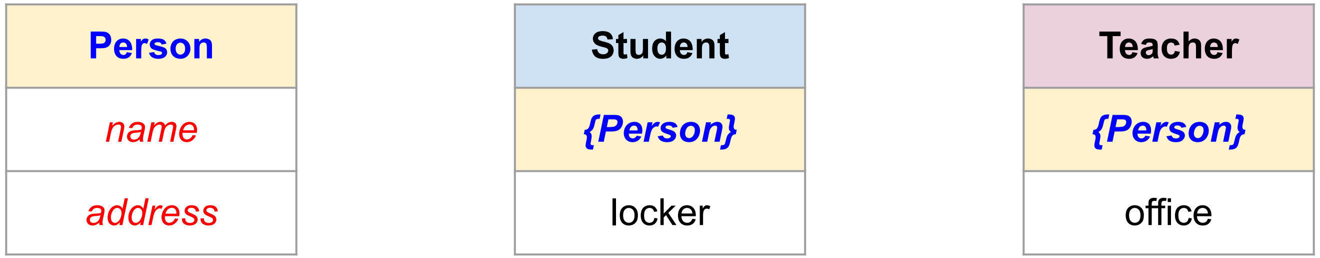
**Inheritance** allows your program to efficiently share common cod
different objects **(code reuse)**; helps you better organize your pro
that model the real world; and create smaller units of maintenance

**Identifying and centralizing common information is called "generalization"**



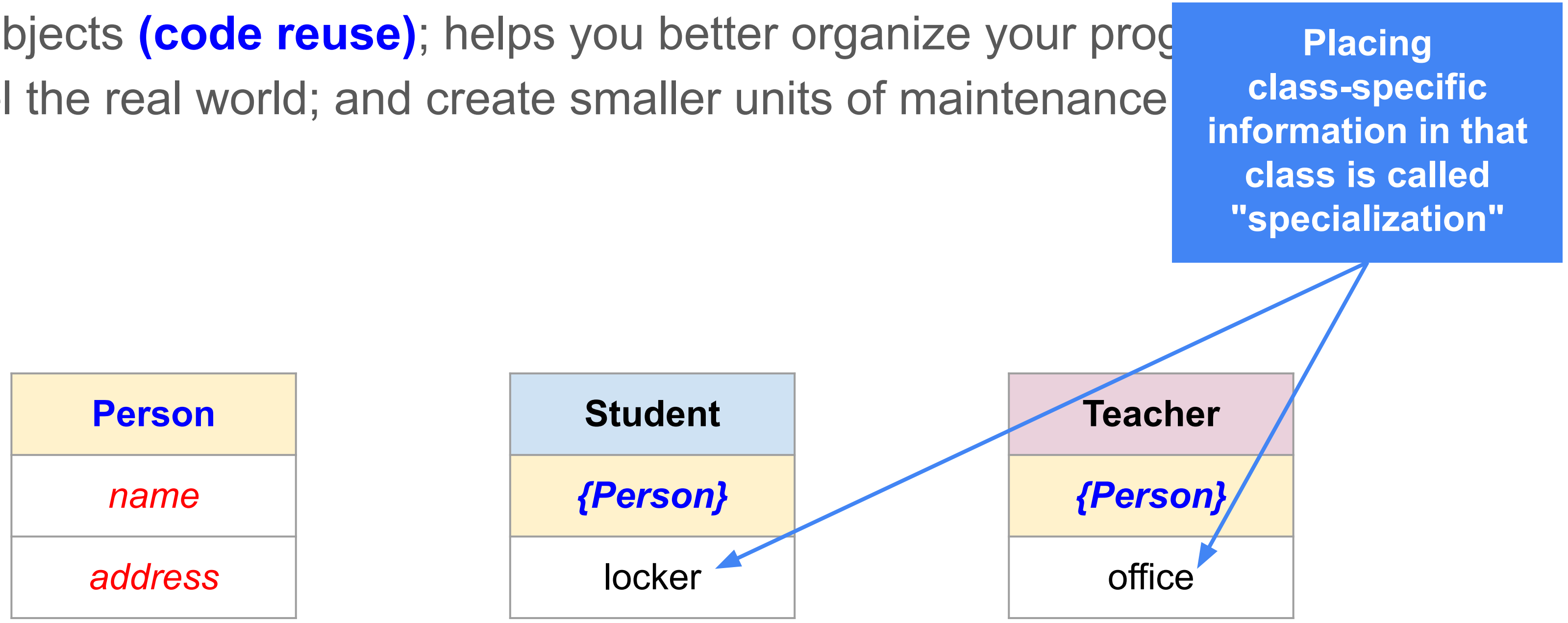
## OOP & Inheritance & Generalization

**Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.



# OOP & Inheritance & Generalization & Specialization

**Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your prog that model the real world; and create smaller units of maintenance



**Placing class-specific information in that class is called "specialization"**
Person
name
address
Student
{Person}
locker
Teacher
{Person}
office

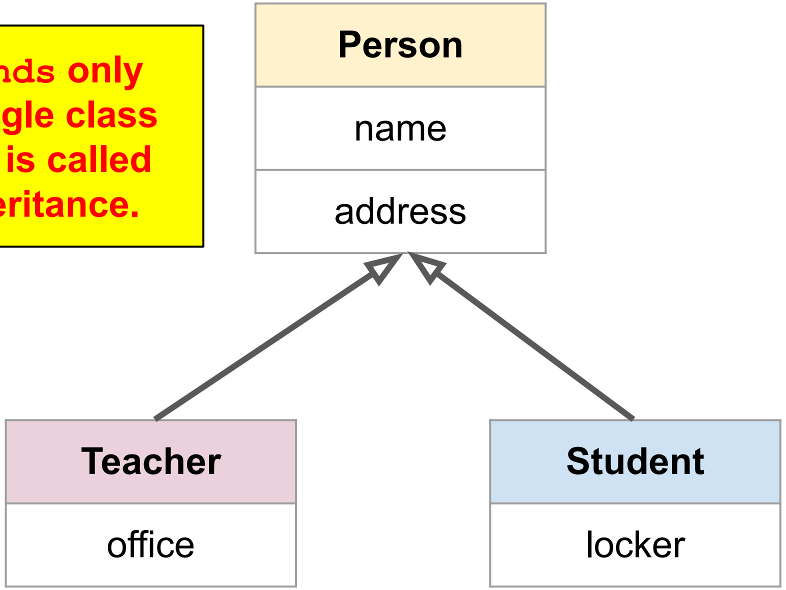
## Java & Inheritance

Any class not marked **`final`** can be used as a superclass.

To be a subclass of something other than Object, use the `extends` keyword.

```
class Person {
  public String name;
  public String address;
}

class Teacher extends Person {
  public String office;
}

class Student extends Person {
  public String locker;
}
```

**Java extends only allows a single class name - this is called single-inheritance.**



## Java & Inheritance

Subclasses **inherit** all the **public** variables and methods of their superclass

```
class Person {
  public String name;
  public String address;
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}

class Student extends Person {
  public String locker;
}
```

```
Person p = new Person();
p.name = "Gary";
p.address = "San Francisco";
p.printInfo();

Teacher t = new Teacher();
t.name = "Chris";
t.address = "San Mateo";
t.printInfo();
t.office = "215W";

Student s = new Student();
s.name = "Beatrice";
s.address = "Colma";
s.printInfo();
s.locker = "B32";
```

## Java & Inheritance

Classes that do not use the `extends` keyword automatically extend the `Object` class (has been happening for every class created since Unit 5)

```
class Object {
  public String toString();
  public boolean equals(Object obj);
  ...
}

class Account {
  public String name;
  public double balance;
}
```

```
Object o = new Object();
o.toString();
o.equals(void);


Account a = new Account();
a.toString();
a.equals(void);
a.name = "Amazon";
a.balance = 0.0;
```

## Java & Inheritance & `is-a` relationships

| Using **Inheritance** results in classes that have `is-a` relationships | |
|---|---|
| `class Account {}` | `Account is-a Object` |
| `class Person {}`<br>`class Teacher extends Person {}`<br>`class Student extends Person {}` | `Person is-a Object`<br>`Teacher is-a Person / Teacher is-a Object`<br>`Student is-a Person / Student is-a Object` |
| `class Animal {}`<br>`class Dog extends Animal {}`<br>`class Snake extends Animal {}` | `Animal is-a Object`<br>`Dog is-a Animal / Dog is-a Object`<br>`Snake is-a Animal / Snake is-a Object` |
| `class Shape {}`<br>`class Square extends Shape {}`<br>`class Circle extends Shape {}`<br>`class Triangle extends Shape {}`<br>`class Pentagon extends Shape {}` | `Shape is-a Object`<br>`Square is-a Shape / Square is-a Object`<br>`Circle is-a Shape / Circle is-a Object`<br>`Triangle is-a Shape / Triangle is-a Object`<br>`Pentagon is-a Shape / Pentagon is-a Object` |

# Java & Inheritance & `is-a` relationships

| The **`instanceof`** operator in Java can be used to test for `is-a` relationships | |
|---|---|
| `class Account {}` | `Account a = new Account();`<br>`System.out.println(a instanceof Object); // true`<br>`System.out.println(a instanceof Account); // true` |
| `class Person {}`<br>`class Teacher extends Person {}` | `Person p = new Person();`<br>`System.out.println(p instanceof Object); // true`<br>`System.out.println(p instanceof Person); // true`<br><br>`Teacher t = new Teacher();`<br>`System.out.println(t instanceof Object); // true`<br>`System.out.println(t instanceof Person); // true`<br>`System.out.println(t instanceof Teacher); // true` |

## Containment & `has-a` relationships

Another concept utilized by Object-Oriented programming languages is **Containment** - where a class is responsible for maintaining an instance of another class inside itself. This results in a `has-a` relationship. We have been using this quite a lot in our examples and projects

```
class Test {
  public String name;
  public double score;
}

class Course {
  public String name;
  public Test tests[10];
}

class Student {
  public String name;
  public Course courses[5];
}
```

```
Test has-a String (name)

Course has-a String (name)
Course has-a Test[] (tests)

Student has-a String (name)
Student has-a Course[] (courses)
```

# Modeling `is-a` & `has-a` Relationships

| | | | `is-a` OR `has-a` |
|---|---|---|---|
| `Pet` | `Cat` | `Dog` | **`Dog is-a Pet`**<br>**`Cat is-a Pet`** |
| `Student` | `Teacher` | `Class` | **`Class has-a Teacher`**<br>**`Class has-a Student`** |
| `Book` | `Movie` | `Media` | **`Movie is-a Media`**<br>**`Book is-a Media`** |
| `Circle` | `Shape` | `Square` | **`Circle is-a Shape`**<br>**`Square is-a Shape`** |
| `Lunch` | `Meal` | `Food` | **`Lunch is-a Meal`**<br>**`Meal has-a Food`** |

# Java & Inheritance

**Note: Not a best practice to make these `public`**

Subclasses can only access the **public** variables and **public** methods of their superclass

```
class Person {
  public String name;
  public String address;
  public void printInfo() {
    String info = buildInfoString();
    System.out.println(info);
  }
  private String buildInfoString() {
    return name + " " + address;
  }
}

class Teacher extends Person {
  public String office;
  public String getBuildInfoString() {
    return buildInfoString();  ** ERROR **
  }
}
```

```
Person p = new Person();
p.name = "Gary";
p.address = "San Francisco";
p.printInfo();



Teacher t = new Teacher();
t.name = "Chris";
t.address = "San Mateo";
t.printInfo();
t.buildInfoString(); ** ERROR **
t.office = "215W";
```

## Java & Inheritance

**Note: This is the better practice - Keep your variables `private` unless you want other code to mess with 'em!**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  ...
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();
```

## Java & Inheritance

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  ...
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();
```

```
error: constructor Person in class Person cannot be applied to given
types;
class Teacher extends Person {
^
  required: String,String
  found:    no arguments
  reason: actual and formal argument lists differ in length
```

## Java & Inheritance

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String addres
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address
  }
}

class Teacher extends Person {
  private String office;
  ...
}
```

**Reminder: If you declare any Constructor, Java will no longer automatically create a no-param constructor for you.**

**In this example, since Person has a Constructor that requires two parameters, there is no way to create a Person with zero parameters.**

**And this error is telling you that Teacher is malformed because there is no way to properly create its Person superclass.**

```
error: constructor Person in class Person cannot be applied to given
types;
class Teacher extends Person {
^
  required: String,String
  found:    no arguments
  reason: actual and formal argument lists differ in length
```

# Java & Inheritance

**Solution 1: Add a no-param constructor to Person**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  ...
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();
```

# Java & Inheritance

**Solution 1: Add a no-param constructor to Person**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person() {
    // empty
  }
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
```

Adding a no-param constructor to Person "fixes" the error - Java now has a way to create a Teacher and its Person superclass.

But did this really fix the issue?

# Java & Inheritance

**Solution 1: Add a no-param constructor to Person**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person() {
    // empty
  }
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Adding a no-param constructor to Person "fixes" the error - Java now has a way to create a Teacher and its Person superclass.

But did this really fix the issue?

Q: What is the output of t.printInfo()?
A: null null

## Java & Inheritance

**Solution 1: Add a no-param constructor to Person**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person() {
    // empty
  }
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Adding a no-param constructor to Person "fixes" the error - Java now has a way to create a Teacher and its Person superclass.

But did this really fix the issue?

Q: What is the output of t.printInfo()?
A: null null

So let's try something else...

## Java & Inheritance

**Solution 2: Call the Person constructor from a Teacher constructor using `super()`**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

# Java & Inheritance

**Solution 2: Call the Person constructor from a Teacher constructor using `super()`**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

**Subclasses can invoke a constructor in their superclass with super()**

**1) super() may only be used on the first line of a subclass constructor**
**2) the params you pass to super() determine which superclass constructor is invoked**

## Java & Inheritance

**Solution 2: Call the Person constructor from a Teacher constructor using `super()`**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

**Subclasses can invoke a constructor in their superclass with super()**

**1) super() may only be used on the first line of a subclass constructor**
**2) the params you pass to super() determine which superclass constructor is invoked**

**Q: Now what is the output of t.printInfo()?**
**A: <a name> <an address>**

## Java & Inheritance

**Solution 2: Call the Person constructor from a Teacher constructor using `super()`**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

```
Subclasses can invoke a constructor in their superclass
with super()

1) super() may only be used on the first line of a
subclass constructor
2) the params you pass to super() determine which
superclass constructor is invoked

Q: Now what is the output of t.printInfo()?
A: <a name> <an address>

Better - But probably still not the best solution...
```

# Java & Inheritance

**Solution 3: Invoke super() from a 2-param Teacher constructor**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher(String name, String address) {
    super(name, address);
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher("Chris", "Thilgen");
t.printInfo();
```

# Java & Inheritance

**Solution 3: Invoke super() from a 2-param Teacher constructor**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher(String name, String address) {
    super(name, address);
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher("Chris", "San Mateo");
t.printInfo();
```

```
Q: Now what is the output of t.printInfo()?
A: Chris San Mateo
```

# Java & Inheritance

**Solution 3: Invoke super() from a 2-param Teacher constructor**

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  private String office;
  public Teacher(String name, String address) {
    super(name, address);
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher("Chris", "San Mateo");
t.printInfo();
```

```
Q: Now what is the output of t.printInfo()?
A: Chris San Mateo

Huzzah!
```

## Java & Inheritance

**Summary**

- Subclasses cannot access `private` variables and methods of a superclass
  Solution: Create accessor methods or carefully use `protected`
- Subclasses can use **`super()`** to invoke superclass constructors
  - **`super()`** can only be used on the first line of a subclass constructor (to prevent subclasses from interfering with the creation of the superclass)
  - The params passed to **`super()`** determine which constructor in the superclass is invoked
  - If you do not add a call to **`super()`** in a subclass constructor, Java will automatically add **`super()`**
    It will call the no-param constructor of the superclass. This ensures that any initialization done by the superclass happens properly.

# Overriding Methods

Method Overriding - To implement a new version of a method to replace code that would otherwise have been inherited from a superclass

To override a method from a superclass, implement the method in the subclass.

```
public class Turtle {
  private int x, y;
  public void forward(int z) {
    x += z;
  }
  public int getX() {
    return x;
  }
  public void setX(int x) {
    this.x = x;
  }
}
```

```
public class TurboTurtle extends Turtle {
  public void forward(int z) {
    setX(getX() + 100*z);
  }
}

// QUESTION: Why did we have to use the
// accessor methods in TurboTurtle instead
// of just saying x += 100*z?
```

# Confuseth them not: Overloading and Overriding

We learned previously about Method Overloading: defining methods with the same name but different method signatures.

When you **overload** a method, you implement a new method with the same name but different parameters.

**Overriding** a method is very different than overloading a method.

***Overriding* replaces an inherited method.**

***Overloading* creates a complementary method with the same name.**

**Overloading**

```
public class MyMath {
  public double sqr(double x) { ... }
  public int sqr(int x) { ... }
}
```

**Overriding**

```
public class MyBetterMath extends MyMath {
  // I have come up with a FASTER,
  // more efficient method to square
  // doubles!
  @Override
  public double sqr(double x) { ... }
}
```

## Failure To Override

To override a method, you must match the method signature exactly. (Parameter names can be different... but the method name, return type, and parameter types must all match.)

If you don't match it up, you may accidentally create an overloaded method! And the compiler will not warn you.

```
class Base {
 public void method() {
   System.out.println("Base.method");
 }
}
class OverrideTest extends Base {
 public void methud() {
   System.out.println("OverrideTest.method");
 }
 public static void main(String args[]) {
   new OverrideTest().method();
 }
}
```

Accidental typo

## @Override

Annotations are additional information that can be specified on Java methods, variables and classes. They start with `@`

`@Override` is an annotation which tells the compiler that you're trying to override a method... and if your declaration doesn't actually override a method, you get a helpful compile error.

(Why an annotation, not a keyword like C#? Adding keywords to an existing language is hard.)

Trying to override: public boolean equals(Object other); in class Object

```
class Rectangle {
 @Override
 public boolean equals(Rectangle other) {
   return other != null &&
          left == other.left &&
          top == other.top &&
          right == other.right &&
          bottom == other.bottom;
 }
}
```

```
Rectangle.java:3: error: method does not
override or implement a method from a
supertype
 @Override
 ^
1 error
```

## The `Object` Superclass

All classes are subclasses of the `Object` class.

(Not necessarily direct subclasses... there are often some superclasses in between.)

When you code, you want to use as specific a type as you can, but this is legal:

```
Object myObject = "Hello, world!";
```

because every String is also an Object.

You also can assign a String to an Object variable, or otherwise use a String in any Object context.

**It is always legal and requires no special syntax to convert a reference to a superclass reference:**

```
String myString = "Hello, world!";
Object myObject = myString;
```

## The `Object` Superclass

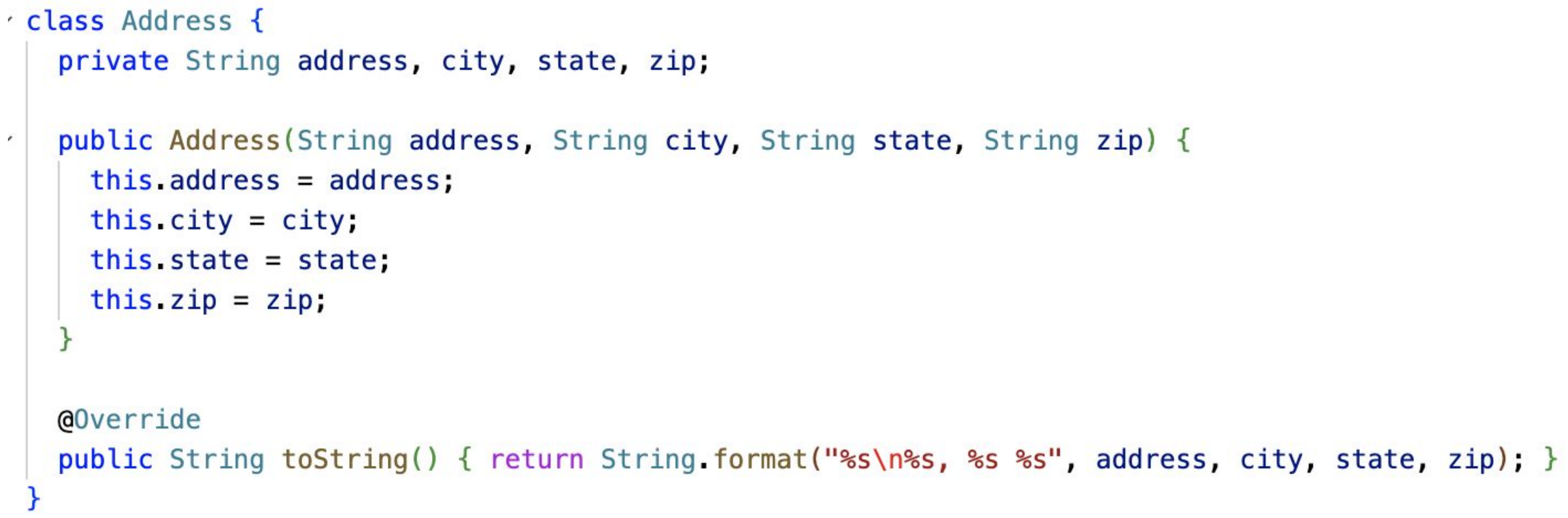
The `Object` class lives in the java.lang package, which is where classes fundamental to the Java language live, like java.lang.String.

`Object` has several public methods. This means that all Java objects inherit these methods. Many of them can be overridden.

Some of the most common ones to override are:

- `public String toString();`
- `public boolean equals(Object o);`

## Overriding the `toString` method



```java
class Address {
  private String address, city, state, zip;

  public Address(String address, String city, String state, String zip) {
    this.address = address;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  @Override
  public String toString() { return String.format("%s\n%s, %s %s", address, city, state, zip); }
}
```

## Overriding the `equals` method

Note that the equals method compares the object with type Object.

So any object can be compared with any other object, of any class, using the equals method!

However, you often want your object to only be equal to objects of the same class, and you may need to compare member variables specific to your class.

**What happens if you don't override Object.equals?** The default implementation is essentially the same as == on reference types: It returns true if the other object is the exact same object instance.

```
@Override
public boolean equals(Object o)
{
   // returns true or false
}
```

Careful:
public boolean equals(MyClass o) is NOT an override of Object.equals...
It would make a separate overloaded method, with no warning/error unless you say @Override!

## Casting

The type cast operator () makes it possible to convert a reference of superclass type (such as Object) to a subclass type (such as Student).

This only works if the superclass reference really IS pointing to an instance of the subclass!

If it isn't, a `ClassCastException` will be thrown.

NOTE: You do NOT need the `(cast)` operator to go from a subclass type to a superclass!

```
Object obj = new String("Hello");
```

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

Bug: This could throw ClassCastException, if one passes, say, a Date or String to this method.

## instanceof operator

The instanceof operator lets you check the "is-a" relationship of an object with a class.

x instanceof T evaluates to true if the object reference x is of type T.

Using instanceof, we can check that the cast to Student is safe before doing it.

An alternative is to try/catch ClassCastException, but that's more expensive.

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Student)) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

## Object.equals contract

Java defines a contract that Object.equals implementations must follow:

**Reflexive:** x.equals(x) is true
**Symmetric:** if x.equal(y) is true, y.equals(x) is true
**Transient:** if x.equals(y) and y.equals(z) are true, x.equals(z) is true
**Consistent:** x.equals(y) should return the same thing if you call it again, if nothing about them changed
**Handles null:** x.equals(null) should return false. (And it shouldn't crash with a NullPointerException!)

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Student)) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

## `super`.*method*`()`

We've seen the `super` keyword used for **constructor chaining**, where a subclass constructor calls a superclass constructor.

The `super` keyword can also be used to invoke a superclass's version of a method, even if the subclass overrides the method.

Often, an overridden method wants to do everything the original method did, but add on some additional behavior.

```
class Person {
  ...
  public void dump(PrintWriter pw) {
    pw.println("Name: " + name);
  }
}
class Teacher extends Person {
  private String classroom;
  ...
  public void dump(PrintWriter pw) {
    super.dump(pw);
    System.out.println("Classroom: " +
                       classroom);
  }
}
```

## super.method() and super() differences

If you don't use `super` in a subclass constructor, Java does it for you. It adds an implicit `super()` to call the no-param constructor of the superclass, if one exists.

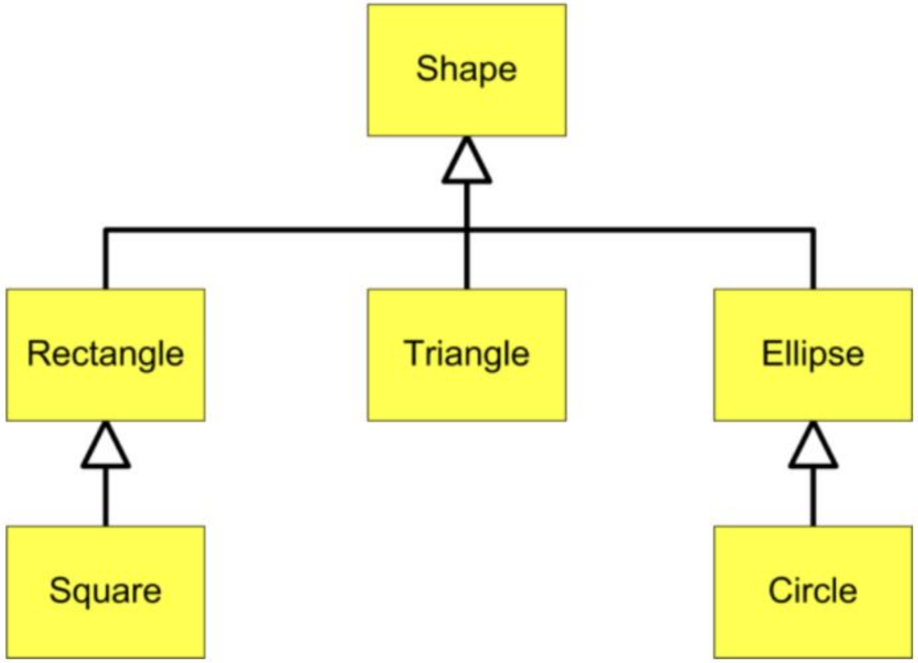
This is done because Java regards constructors as important to making a properly initialized object. You can't skip around a superclass constructor.

Method overrides are different. Java lets you completely replace the definition of a method. The new method code has a choice: It can use `super.`*method*`()` to call the superclass version of the method at some point, or not.

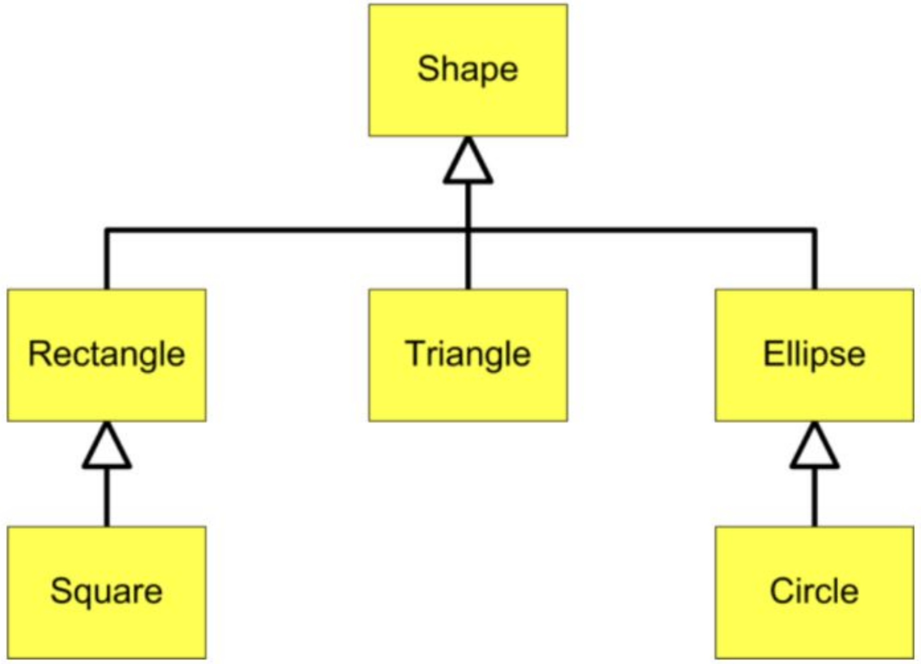
```
class NPC {
  ...
  public void tick() {
    // Implements random movement of the NPC
    // around the map
  }
}

class Teacher extends NPC {
  ...
  public void tick() {
    // call super.tick() to get the basic NPC behavior
    // like random movement
    super.tick();

    // Additional code here for special NPC behavior
    // specific to this character
  }
}
```

## Inheritance Hierarchy

When you use multiple layers of inheritance, this creates a set of relationships called an **inheritance hierarchy** – most often illustrated as a tree.



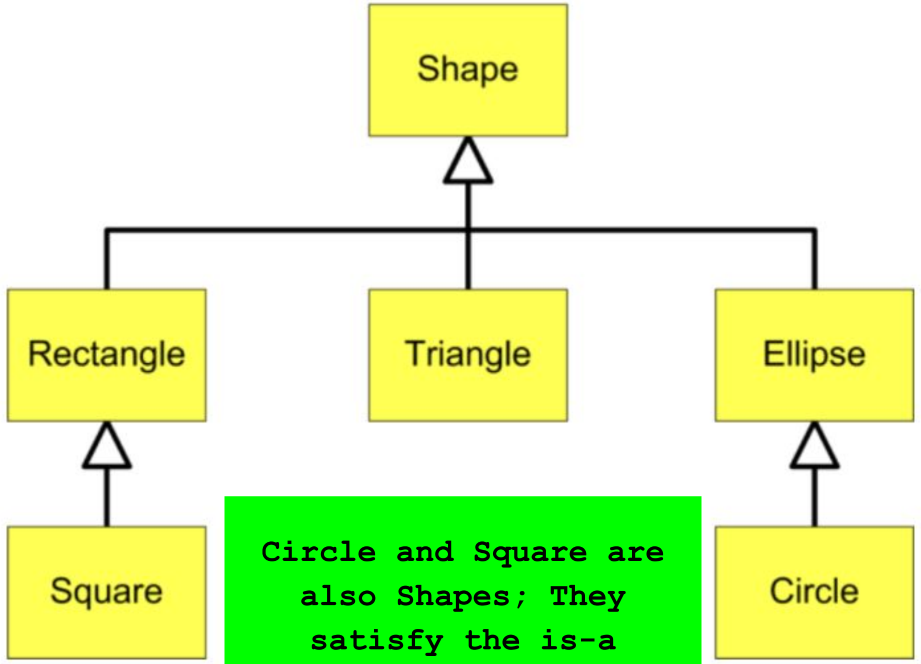
## Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle

Rectangle is-a Shape
```

## Inheritance Hierarchy

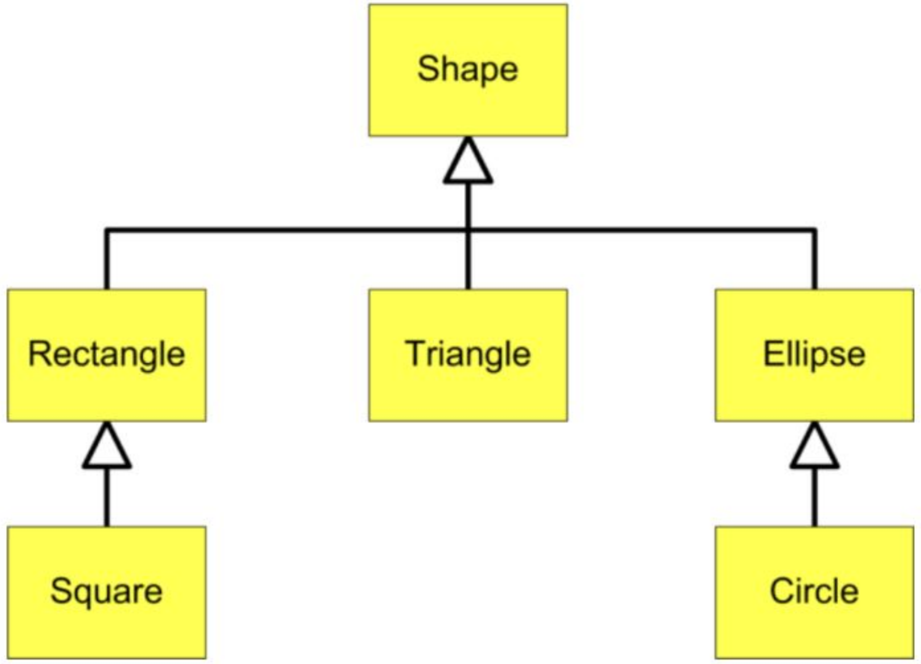


**Circle and Square are also Shapes; They satisfy the is-a relationship test with Shape**

- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle

Rectangle is-a Shape
```

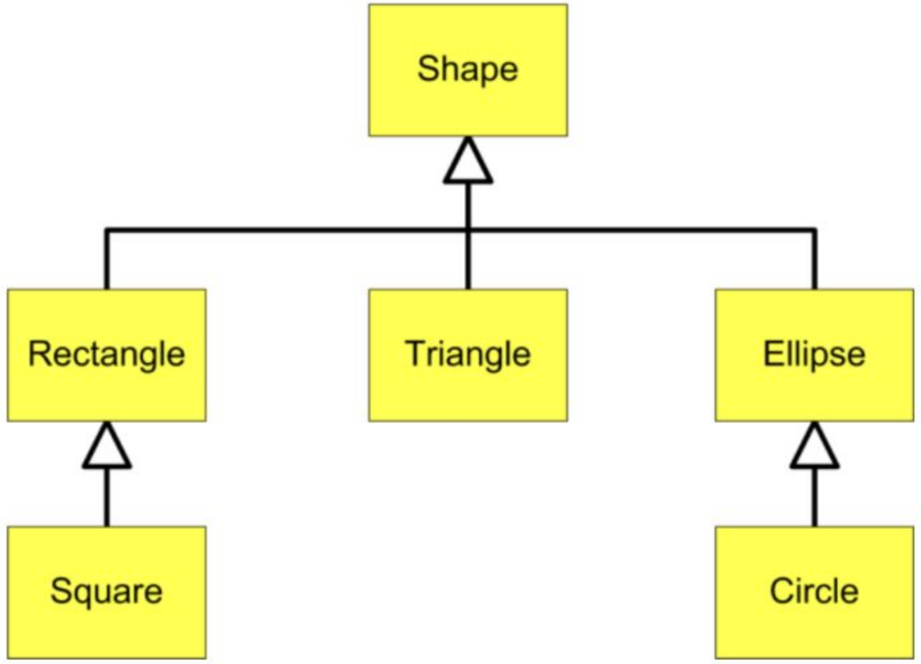
## Inheritance Hierarchy



The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of objects.

```
Circle c = new Circle();
Ellipse e = c;
Shape s = c;
```

## Inheritance Hierarchy

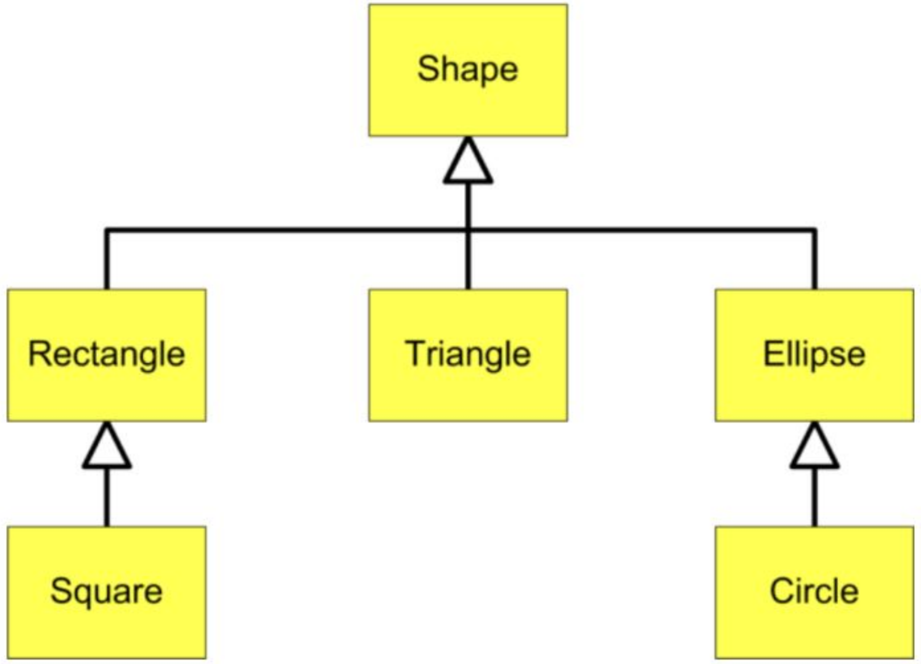


The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of objects.

```
Circle c = new Circle();
Ellipse e = c;
Shape s = c;
```

**This means you can create an `Array/ArrayList` of `Shapes` and add any of these objects to it.**

## Inheritance Hierarchy



- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape

Shape s = new Shape()
Rectangle r = s; **ERROR**
Triangle t = s; **ERROR**
Ellipse e = e; **ERROR**
```

# Compile Time, Run Time

- Java is a **compiled language**. (Unlike Python, which is **interpreted**.)
  - A program must be compiled using the Java compiler, that is, turned into 1's and 0's that the Java Virtual Machine understands.
  - The Java Virtual Machine then can execute the resulting **Java bytecode**.
- The Java compiler reads your Java source code (.java files)
  - Compilation checks your program for syntax errors (missing semicolons, curly braces, etc.) as well as semantic errors (incompatible types, etc.) This produces **compile errors.**
  - This is the program that yells at you if you try to use a variable before you've initialized it.
- But you can also encounter **run-time errors** (usually Java exceptions)
  - For example, an error message that says you tried to divide by zero
  - These kinds of error can't be identified a priori–your code needs to be run for these issues to be caught

*We had these reversed in the 9.6 slides. They have been updated.*

## Polymorphism

In Java, an object variable has both a **declared type** (compile-time type) and an **actual type** (run-time type).

The **declared type** is the type that is used in the declaration.

The **actual type** is the class that was actually used to create the object using `new`.

```java
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```java
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

# Polymorphism

In Java, an object variable has both a **declared type** (compile-time type) and an **actual type** (run-time type).

The **declared type** is the type that is used in the declaration.

The **actual type** is the class that was actually used to create the object using `new`.

```
class Shape { ... }
class Rectangle extends Shape { ... }
```

```
Shape shape = new Rectangle();
```

Declared Type
(Compile-Time Type)

*What the compiler is doing its type checking against, e.g., when it is determining what methods are available to call.*

Actual Type
(Run-Time Type)

*What the object actually is at run-time, which dictates how it will actually behave when methods are invoked.*

## Polymorphism

In this example, the **declared type** of `shapes[i]` is `Shape`.

The **actual types**, however, are `Rectangle`, `Triangle`, and `Ellipse`.

```java
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```java
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

## Polymorphism

When you use `new`, an instance of a specific type is created. This instance will always have that **actual type**, even if you put it in a variable with a **declared type** of a superclass.

**The elements of `shapes` have declared type of `Shape`, but actual types of `Rectangle`, `Triangle`, and `Ellipse` (because that is the type that was created with `new`)**

```
class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

```
> class Rectangle
> class Triangle
> class Ellipse
```

## Polymorphism

The actual type of an object is whatever class was used with the `new` operator.

That object could be referenced by a variable of a superclass type. The declared type is used by the compiler to determine the available methods. The method called, though, depends on the actual type of the object. The object's class may override that method.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
```

**We can invoke `s.draw()` because `draw` is a method of Shape. The actual `draw` method that is invoked depends on the actual type of each object. `Rectangle`, `Triangle`, and `Ellipse` may override `draw`.**

```
class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}

> class Rectangle
> class Triangle
> class Ellipse
```

## Polymorphism

### At **compile-time**

- The compiler uses the **declared type / compile-time type** to verify that the methods you are trying to use are available to an object of that type.
- The code won't compile if the methods don't exist in that class or some superclass.

```
Shape s = new Rectangle();
```

Because `s` is *declared* as Shape, only `Shape` methods can be invoked on s, even though it's really a `Rectangle`!

### At **run-time**

- The Java Virtual Machine uses the **run-time type** to determine which methods are used
- When a method is called, the JVM checks whether the run-time type overrides the method, and runs the overridden version of the method if so.

## Polymorphism

- **Polymorphic Assignment**
  - `Shape s = new Rectangle();`
- **Polymorphic Parameters**
  - `public void print(Shape s){}`
- **Polymorphic Collections**
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

## Polymorphism

- **Polymorphic Assignment**
  - `Shape s = new Rectangle();`
- **Polymorphic Parameters**
  - `public void print(Shape s){}`
- **Polymorphic Collections**
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

**There are no errors at compile-time because the compiler checks that the “subclass is-a superclass” relationship is true.**

**At run-time, the Java runtime will use the object’s actual subclass type and call the subclass methods for any overridden methods.**

**This is why they are polymorphic – the same code can have different results depending on the object’s actual type at run-time.**

## A note on the AP Java Subset

## What IS the AP Java Subset?

Java is an enormous programming language.

It's been around for 30 years, and many language features have been added over that time, and the Java standard library has grown and grown.

The AP exam defines a subset of language features and the Java standard library that you are actually tested on. For the standard library, you get a one-page quick reference sheet on the classes and methods you may be expected to use.

What is in the AP Java Subset exactly?

Hard to find official word... This College Board PDF may be accurate, but it's from 2014.

I think the way to think about it is, if it's in CS Awesome, it's in the AP Java Subset. Otherwise, not.

Mostly we have followed CS Awesome in class, but there are a few features we covered that are not in the AP Java Subset.

## Bonus material not in the AP Java Subset

These are "bonus" topics which we covered or used that you do not need to know for the AP Exam:

- `abstract` classes and methods
- Interfaces
- `protected` access modifier
- `this()` constructor chaining
- Control flow: `switch`, `break`, `continue`, `do...while`
- Linked lists (this used to be part of the now defunct AP Comp Sci AB exam)
- Lots of library stuff: `FileInputStream`/`FileOutputStream`, Java Swing, etc.

Good stuff to know... but you don't need to know it on May 3, 2023!

## On the AP, can students use any feature of Java?

Teachers and students often ask whether one can use any class or method from the Java API, including the latest Java features, in the AP free-response questions. This is a paraphrase of advice from Professor James K. Huggins, a veteran AP CS exam reader:

**Short Answer:** Yes, students can use any valid Java code... but it's risky.

**Long Answer:** If your answer is a valid Java solution to the question, it's eligible for full credit. But...

- Follow any rules stated in the question. If the rules say not to use something, don't use it. If the rules say to use something specific, use that thing.
- The only reference available to you in the exam is the AP Java Subset Quick Reference Sheet. If you use something from outside the subset and mis-remember it, you'll lose points.
- If you use a fancy feature outside the AP Java Subset, the readers are only human and don't have 100% of the Java standard library memorized. They may not understand your code, and dock points.

So, what he tells his students is:

- The questions are designed to be solved within the boundaries of the AP Java Subset. There's no need to bring in outside features.
- If you can't see an "in-bounds" solution, and you do know an "out-of-bounds" solution, go ahead and write it down.
- Don't try to show off! The AP exam isn't the time to use obscure features of Java, and doing so is risky. Any question on the exam, you should be able to solve with the material you learned in this course.

# AP CS FRQ 2

(25 minutes)

2022 AP Computer Science A - Free-Response Questions

**Complete (2)**

# AP CS FRQ 2 - Review

(15 minutes)

[Sample Responses and Scoring Commentary - FRQ-2]()

Spoilers Ahead

```
class Textbook extends Book {
  // Don't forget to declare this! And it MUST be private!
  private int edition;
  public Textbook(String bookTitle, double bookPrice, int edition) {
    // Remember super() is required here, because Book has no no-param constructor.
    super(bookTitle, bookPrice);
    this.edition = edition;
  }
  // Remember you cannot reference "title" directly which is private! And use equals()!
  // You can say other.edition, but I used other.getEdition() anyway.
  public boolean canSubstituteFor(Textbook other) {
    return getTitle().equals(other.getTitle()) && edition >= other.getEdition();
  }
  // They want you to use super.method() here; don't duplicate Book's logic.
  public String getBookInfo() {
    return super.getBookInfo() + "-" + edition;
  }
  // Make sure you remember to write this method.
  public int getEdition() { return edition; }
}
```