Classes and Objects

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

CHEN-WEI WANG

Current slides are cross-referenced throughout this review
tutorials on Java OOP:

https://www.eecs.yorku.ca/~jackie/teaching/
tutorials/index.html#refurbished_store

Optional: Tutorial Videos to Help You Review

- **Link to Tutorial Series**:
  `https://www.eecs.yorku.ca/~jackie/teaching/`
  `tutorials/index.html#java_from_scratch_w21`
  - **Week 1**: **Eclipse** work environment
  - *Week 2c, 2d, 2e*: **Debugger** in Eclipse
  - **Weeks 2, 3**: Programming/Debugging **Conditionals**
  - **Weeks 4, 5**: Programming/Debugging **Arrays** and **Loops**
  - *Weeks 6, 7, 8*: **Classes** and **Objects**
- **iPad Notes**: `https://www.eecs.yorku.ca/~jackie/`
  `teaching/tutorials/notes/EECS1022%20Tutorial%`
  `20on%20Java.pdf`

Required: Written Notes to Review

- **Inferring Classes/Methods from JUnit Tests**:
  `https://www.eecs.yorku.ca/~jackie/teaching/`
  `lectures/2021/F/EECS2030/notes/EECS2030 F21`
  `Inferring Classes from JUnit.pdf`

- **Declaring and Manipulating Reference-Typed, Multi-Valued Attributes**: `https://www.eecs.yorku.ca/~jackie/`
  `teaching/lectures/2021/F/EECS2030/notes/`
  `EECS2030 F21 Tracing PointCollectorTester.pdf`

Learning Outcomes

Understand:
- Object Orientation
- Classes as <u>Templates</u>:
  - attributes, constructors, (accessor and mutator) methods
  - use of `this`
- Objects as <u>Instances</u>:
  - use of `new`
  - the dot notation, method invocations
  - reference aliasing
- Reference-Typed Attributes: Single-Valued vs. Multi-Valued
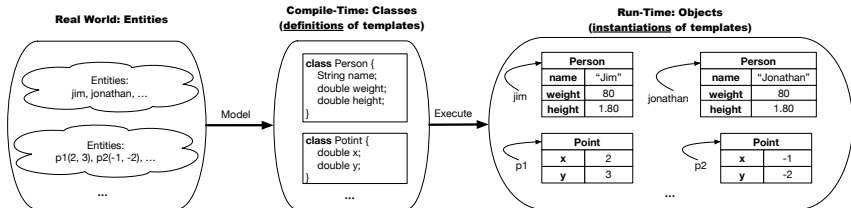- Non-Static vs. Static Variables
- Helper Methods

- In EECS1022/EECS1021:
  - *Model Component*: One or More Java Classes

    e.g., `Person` vs. `SMS`, `Student`, `CourseRecord`
  - Another Java class that "manipulates" the model class(es)
    - *Controller* (e.g., `BMIActivity`, `LEDController`). Effects?
      Visualized at a connected physical device (e.g., tablet, LED lightbulbs)
    - *Tester* (e.g., `PersonTester`, `BankTester`). Effects?
      Seen (as textual outputs) at console
      Asserting *expected* vs. *actual* Values in JUnit tests
- In Java:
  - We may define more than one *classes*.
  - Each class may contain more than one *methods*.

  Object-Oriented Programming (OOP) in Java:

  - Use *classes* to define templates
  - Use *objects* to instantiate classes
  - At *runtime*, *create* objects and *call* methods on objects, to *simulate interactions* between real-life entities.

Object Orientation:
Observe, Model, and Execute

Real World: Entities → Model → Compile-Time: Classes (**definitions** of templates) → Execute → Run-Time: Objects (**instantiations** of templates)

- Study this tutorial video that walks you through the idea of *object orientation* .
- We *observe* how real-world *entities* behave.
- We *model* the common *attributes* and *behaviour* of a set of entities in a single *class*.
- We *execute* the program by creating *instances* of classes, which interact in a way analogous to that of real-world *entities*.

Object-Oriented Programming (OOP)

- In real life, lots of *entities* exist and interact with each other.
  - e.g., *People* gain/lose weight, marry/divorce, or get older.
  - e.g., *Cars* move from one point to another.
  - e.g., *Clients* initiate transactions with banks.
- Entities:
  - Possess *attributes*;
  - Exhibit *bebaviour*; and
  - Interact with each other.
- Goals: Solve problems *programmatically* by
  - *Classifying* entities of interest
    Entities in the same class share *common* attributes and bebaviour.
  - *Manipulating* data that represent these entities
    Each entity is represented by *specific* values.

Points on a two-dimensional plane are identified by their signed distances from the X- and Y-axises. A point may move arbitrarily towards any direction on the plane. Given two points, we are often interested in knowing the distance between them.

- A template called `Point` defines the common
  - ○ *attributes* (e.g., `x`, `y`)   [≈ nouns]
  - ○ *behaviour* (e.g., move up, get distance from)   [≈ verbs]

OO Thinking: Templates vs. Instances (1.2)

- A *template* (e.g., class `Point`) defines what's **shared** by a set of related entities (i.e., 2-D points).
  - Common *attributes* (x, y)
  - Common *behaviour* (move left, move up)
- Each template may be *instantiated* as multiple instances, each with *instance-specific* values for attributes x and y:
  - `Point` instance `p1` is located at $(3, 4)$
  - `Point` instance `p2` is located at $(-4, -3)$
- Instances of the same template may exhibit *distinct behaviour*.
  - When `p1` moves up for 1 unit, it will end up being at $(3, 5)$
  - When `p2` moves up for 1 unit, it will end up being at $(-4, -2)$
  - Then, `p1`'s distance from origin: $[\sqrt{3^2 + 5^2}]$
  - Then, `p2`'s distance from origin: $[\sqrt{(-4)^2 + (-2)^2}]$

A person is a being, such as a human, that has certain
attributes and behaviour constituting personhood: a person
ages and grows on their heights and weights.

- A template called `Person` defines the common
  - *attributes* (e.g., `age`, `weight`, `height`)   [≈ nouns]
  - *behaviour* (e.g., get older, gain weight)   [≈ verbs]

OO Thinking: Templates vs. Instances (2.2)

- A *template* (e.g., class `Person`) defines what's **shared** by a set of related entities (i.e., persons).
  - Common *attributes* (`age`, `weight`, `height`)
  - Common *behaviour* (get older, lose weight, grow taller)
- Each template may be *instantiated* as multiple instances, each with *instance-specific* values for attributes `age`, `weight`, and `height`.
  - `Person` instance `jim` is
    50-years old, 1.8-meters tall and 80-kg heavy
  - `Person` instance `jonathan` is
    65-years old, 1.73-meters tall and 90-kg heavy
- Instances of the same template may exhibit *distinct behaviour*.
  - When `jim` gets older, he becomes 51
  - When `jonathan` gets older, he becomes 66.
  - `jim`'s BMI is based on his own height and weight $[\frac{80}{1.8^2}]$
  - `jonathan`'s BMI is based on his own height and weight $[\frac{90}{1.73^2}]$

In Java, you use a **class** to define a *template* that enumerates *attributes* that are common to a set of *entities* of interest.

```java
public class Person {
  private int age;
  private String nationality;
  private double weight;
  private double height;
}
```

```java
public class Point {
  private double x;
  private double y;
}
```

Java Data Types (1)

A (data) type denotes a set of related *runtime values*.

1. *Primitive Types*
    ○ *Integer* Type
        • int   [set of 32-bit integers]
        • long   [set of 64-bit integers]
    ○ *Floating-Point Number* Type
        • double   [set of 64-bit FP numbers]
    ○ *Character* Type
        • char   [set of single characters]
    ○ *Boolean* Type
        • boolean   [set of true and false]
2. *Reference Type* : *Complex Type with Attributes and Methods*
    ○ *String*   [set of references to character sequences]
    ○ *Person*   [set of references to Person objects]
    ○ *Point*   [set of references to Point objects]
    ○ *Scanner*   [set of references to Scanner objects]

Java Data Types (2)

- A variable that is declared with a *type* but *uninitialized* is implicitly assigned with its **default value**.
  - **Primitive Type**
    - `int i;`   [**0** is implicitly assigned to `i`]
    - `double d;`   [**0.0** is implicitly assigned to `d`]
    - `boolean b;`   [**false** is implicitly assigned to `b`]
  - **Reference Type**
    - `String s;`   [**null** is implicitly assigned to `s`]
    - `Person jim;`   [**null** is implicitly assigned to `jim`]
    - `Point p1;`   [**null** is implicitly assigned to `p1`]
    - `Scanner input;`   [**null** is implicitly assigned to `input`]
- You *can* use a <u>primitive</u> variable that is *uninitialized*.
  Make sure the **default value** is what you want!
- Calling a method on a *uninitialized* <u>reference</u> variable crashes your program.   [*NullPointerException*]
  Always initialize <u>reference</u> variables!

OOP: Methods (1.1)

- A **method** is a named block of code, *reusable* via its name.

$T_1\ p_1$

$T_2\ p_2$

$\ldots$

$T_n\ p_n$

m

{
…
/* implementation of method m */
}

RT
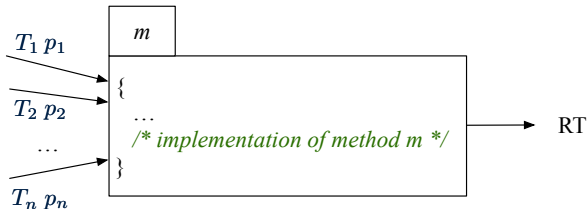
- The *Header* of a method consists of:
  - Return type   [*RT* (which can be `void`)]
  - Name of method   [*m*]
  - Zero or more *parameter names*   [$p_1, p_2, \ldots, p_n$]
  - The corresponding *parameter types*   [$T_1, T_2, \ldots, T_n$]
- A call to method *m* has the form: $m(a_1, a_2, \ldots, a_n)$
  Types of *argument values* $a_1, a_2, \ldots, a_n$ must match the the corresponding parameter types $T_1, T_2, \ldots, T_n$.

OOP: Methods (1.2)

- In the body of the method, you may
  - Declare new *local variables* (whose **scope** is within that method).
  - Use or change values of *attributes*.
  - Use values of *parameters*, if any.

```
public class Person {
  private String nationality;
  public void changeNationality(String newNationality) {
    nationality = newNationality; } }
```

- *Call* a *method*, with a **context object**, by passing *arguments*.

```
public class PersonTester {
  public static void main(String[] args) {
    Person jim = new Person(50, "British");
    Person jonathan = new Person(60, "Canadian");
    jim.changeNationality("Korean");
    jonathan.changeNationality("Korean"); } }
```

OOP: Methods (2)

- Each **class** C defines a list of methods.
  - A **method** m is a named block of code.

- We *reuse* the code of method m by calling it on an **object** obj of class C.

  For each **method call** obj.m(...):
  - obj is the **context object** of type C
  - m is a method defined in class C
  - We intend to apply the *code effect of method* m to object obj.

    e.g., jim.getOlder() vs. jonathan.getOlder()

    e.g., p1.moveUp(3) vs. p2.moveUp(3)

- All objects of class C share *the same definition* of method m.

- However:
  - ∵ Each object may have *distinct attribute values*.
  - ∴ Applying *the same definition* of method m has *distinct effects*.

OOP: Methods (3)

1. *Constructor*
   - Same name as the class. No return type. *Initializes* attributes.
   - Called with the **new** keyword.
   - e.g., `Person jim = new Person(50, "British");`

2. *Mutator*
   - *Changes* (re-assigns) attributes
   - `void` return type
   - Cannot be used when a value is expected
   - e.g., `double h = jim.setHeight(78.5)` is illegal!

3. *Accessor*
   - *Uses* attributes for computations (without changing their values)
   - Any return type other than `void`
   - An explicit `return` *statement* (typically at the end of the method) returns the computation result to where the method is being used.
     e.g., `double bmi = jim.getBMI();`
     e.g., `println(p1.getDistanceFromOrigin());`

OOP: Class Constructors (1.1)

- The purpose of defining a *class* is to be able to create *instances* out of it.
- To *instantiate* a class, we use one of its **constructors** .
- A constructor
  - declares input *parameters*
  - uses input parameters to *initialize* **some or all** of its *attributes*

For each *class*, you may define *one or more* *constructors* :

○ *Names* of all constructors must match the class name.
○ *No return types* need to be specified for constructors.
○ **Overloaded** constructor have *distinct* lists of *parameter types*.

  • Person(String n), Person(String n, int age)   ✓
  • Person(String n, int age), Person(int age, String n)   ✓
  • Person(String fN, int age), Person(String lN, int id)   ×

○ Each *parameter* that is used to initialize an attribute must have a *matching type*.
○ The *body* of each constructor specifies how *some or all attributes* may be *initialized*.

OOP: Class Constructors (2.1)

```
public class Point {
  private double x;
  private double y;

  public Point(double initX, double initY) {
    x = initX;
    y = initY;
  }

  public Point(char axis, double distance) {
    if (axis == 'x') { x = distance; }
    else if (axis == 'y') { y = distance; }
    else { /* Error: invalid axis */ }
  }
}
```

OOP: Class Constructors (2.2)

```java
public class Person {
  private int age;
  private String nationality;
  private double weight;
  private double height;
  public Person(int initAge, String initNat) {
    age = initAge;
    nationality = initNat;
  }
  public Person (double initW, double initH) {
    weight = initW;
    height = initH;
  }
  public Person(int initAge, String initNat,
          double initW, double initH) {
    ... /* initialize all attributes using the parameters */
  }
}
```

Visualizing Objects at Runtime (1)

- To trace a program with sophisticated manipulations of objects, it's critical for you to visualize how objects are:
    - Created using *constructors*
      ```
      Person jim = new Person(50, "British", 80, 1.8);
      ```
    - Inquired using *accessor methods*
      ```
      double bmi = jim.getBMI();
      ```
    - Modified using *mutator methods*
      ```
      jim.gainWeightBy(10);
      ```
- To visualize an object:
    - Draw a rectangle box to represent *contents* of that object:
        - Title indicates the *name of class* from which the object is instantiated.
        - Left column enumerates *names of attributes* of the instantiated class.
        - Right column fills in *values* of the corresponding attributes.
    - Draw arrow(s) for *variable(s)* that store the object's *address*.

Visualizing Objects at Runtime (2.1)

After calling a *constructor* to create an object:

```
Person jim = new Person(50, "British", 80, 1.8);
```

Person	
age	50
nationality	"British"
weight	80
height	1.8

jim
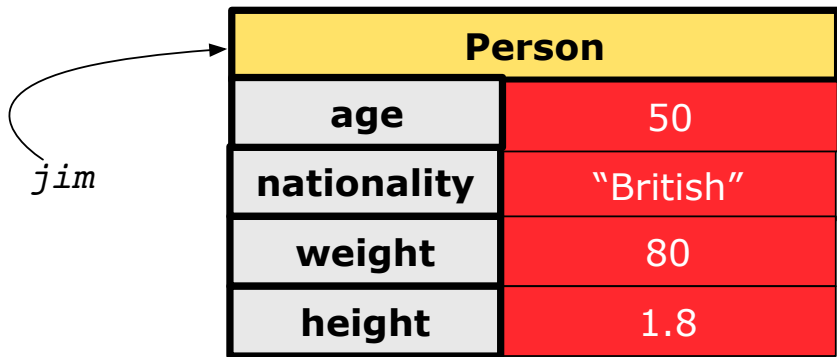
Visualizing Objects at Runtime (2.2)

After calling an *accessor* to inquire about context object `jim`:

```
double bmi = jim.getBMI();
```

- Contents of the object pointed to by `jim` remain intact.
- Retuned value $\frac{80}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.

Person	
age	50
nationality	"British"
weight	80
height	1.8

jim
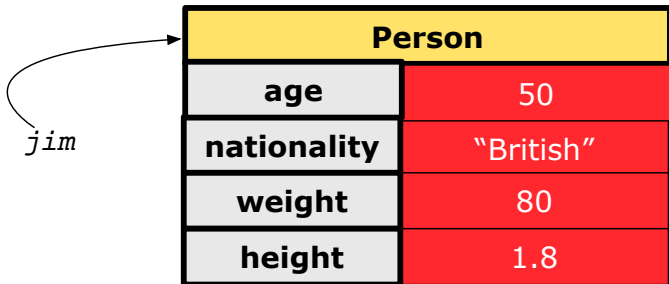
Visualizing Objects at Runtime (2.3)

After calling a *mutator* to modify the state of context object `jim`:

```
jim.gainWeightBy(10);
```

- *Contents* of the object pointed to by `jim` change.
- **Address** of the object remains unchanged.
  ⇒ `jim` points to the same object!

Person	
age	50
nationality	"British"
weight	~~80~~ 90
height	1.8

jim
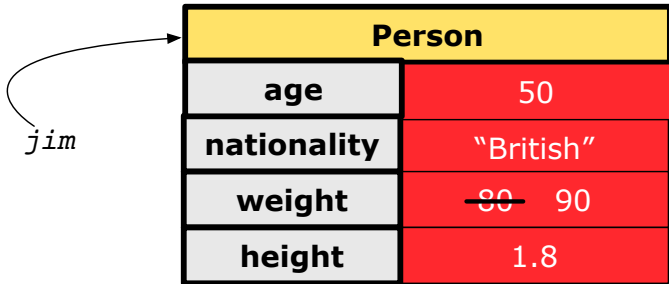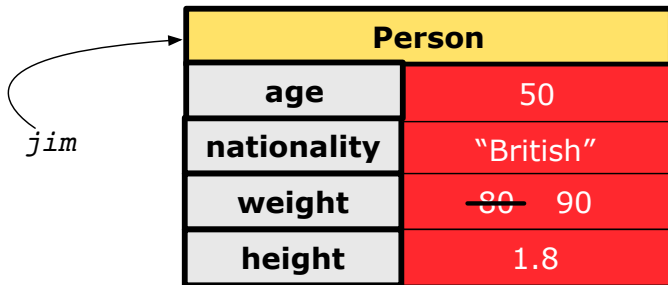
After calling the same *accessor* to inquire the *modified* state of context object `jim`:

```
bmi = jim.getBMI();
```

- Contents of the object pointed to by `jim` remain intact.
- Retuned value $\frac{90}{(1.8)^2}$ of `jim.getBMI()` stored in variable `bmi`.

Person	
age	50
nationality	"British"
weight	~~80~~ 90
height	1.8

jim

Object Creation (1.1)

```
Point p1 = new Point(2, 4);
```

1. **RHS (Source) of Assignment**: `new Point(2, 4)` creates a new *Point object* in memory.

Point	
x	2.0
y	4.0

2. **LHS (Target) of Assignment**: `Point p1` declares a *variable* that is meant to store the *address* of *some Point object*.

3. **Assignment**: Executing `=` stores new object's address in `p1`.

Point	
x	2.0
y	4.0

p1
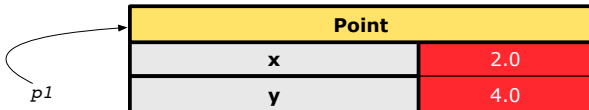
Object Creation (1.2)

```
Person jim = new Person(50, "British");
```

1. RHS (Source) of Assignment: `new Person(50, "British")`
   creates a new *Person object* in memory.

Person	
age	50
nationality	"British"
weight	0.0
height	0.0

2. LHS (Target) of Assignment: `Point jim` declares a *variable*
   that is meant to store the *address* of *some Person object*.

3. Assignment: Executing `=` stores new object's address in `jim`.

Person	
age	50
nationality	"British"
weight	0.0
height	0.0

jim
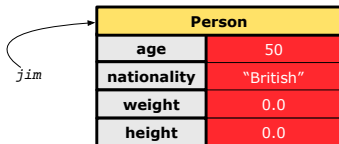
Object Creation (2)

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
Point@677327b6
```

By default, the address stored in `p1` gets printed.

Instead, print out attributes separately:

```
System.out.println("(" + p1.getX()+", "+p1.getY() + ")");
```
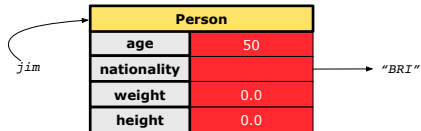
```
(2.0, 4.0)
```

OOP: Object Creation (3.1.1)

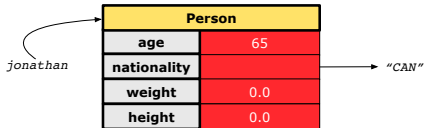A constructor may only *initialize* some attributes and leave others *uninitialized*.

```
public class PointTester {
  public static void main(String[] args) {
    Point p1 = new Point(3, 4);
    Point p2 = new Point(-3 -2);
    Point p3 = new Point('x', 5);
    Point p4 = new Point('y', -7);
  }
}
```
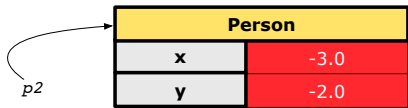
OOP: Object Creation (3.1.2)

Point p1 = **new** Point(3, 4)

Person	
x	3.0
y	4.0

p1

Point p2 = **new** Point(-3, -2)

Person	
x	-3.0
y	-2.0

p2

Point p3 = **new** Point('x', 5)

Person	
x	5.0
y	0

p3

Point p4 = **new** Point('y', -7)

Person	
x	0
y	-7.0

p4
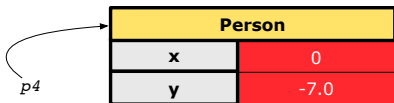
OOP: Object Creation (3.2.1)

A constructor may only *initialize* some attributes and leave others *uninitialized*.

```java
public class PersonTester {
 public static void main(String[] args) {
   /* initialize age and nationality only */
   Person jim = new Person(50, "BRI");
   /* initialize age and nationality only */
   Person jonathan = new Person(65, "CAN");
   /* initialize weight and height only */
   Person alan = new Person(75, 1.80);
   /* initialize all attributes of a person */
   Person mark = new Person(40, "CAN", 69, 1.78);
 }
}
```

OOP: Object Creation (3.2.2)

`Person jim = new Person(50, "BRI")`

Person	
age	50
nationality	
weight	0.0
height	0.0

jim → → "BRI"

`Person jonathan = new Person(65, "CAN")`

Person	
age	65
nationality	
weight	0.0
height	0.0

jonathan → → "CAN"

`Person alan = new Person(75, 1.80)`

Person	
age	0
nationality	
weight	75.0
height	1.80

alan → → null

`Person mark = new Person(40, "CAN", 69, 1.78)`

Person	
age	40
nationality	
weight	69.0
height	1.78

mark → → "CAN"
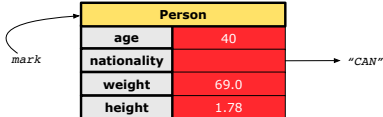
OOP: Object Creation (4)

- When using the constructor, pass *valid* *argument values*:
  - The type of each argument value must match the corresponding parameter type.
  - e.g., `Person(50, "BRI")` matches
    `Person(int initAge, String initNationality)`
  - e.g., `Point(3, 4)` matches
    `Point(double initX, double initY)`
- When creating an instance, *uninitialized* attributes implicitly get assigned the *default values*.
  - Set *uninitialized* attributes properly later using **mutator** methods

```
Person jim = new Person(50, "British");
jim.setWeight(85);
jim.setHeight(1.81);
```

OOP: The Dot Notation (1)

- A binary operator:
  - **LHS** an object
  - **RHS** an attribute or a method
- Given a *variable* of some *reference type* that is **not** `null`:
  - We use a dot to retrieve any of its *attributes* .
    Analogous to 's in English
    e.g., `jim.nationality` means `jim`'s nationality
  - We use a dot to invoke any of its *mutator methods* , in order to
    change values of its attributes.
    e.g., `jim.changeNationality("CAN")` changes the
    `nationality` attribute of `jim`
  - We use a dot to invoke any of its *accessor methods* , in order to
    use the result of some computation on its attribute values.
    e.g., `jim.getBMI()` computes and returns the BMI calculated
    based on `jim`'s weight and height
  - Return value of an *accessor method* must be stored in a variable.
    e.g., `double jimBMI = jim.getBMI()`

The `this` Reference (1)

- Each *class* may be instantiated to multiple *objects* at runtime.

```
public class Point {
  private double x; private double y;
  public void moveUp(double units) { y += units; }
}
```

- Each time when we call a method of some class, using the dot notation, there is a specific *target*/*context* object.

```
1  Point p1 = new Point(2, 3);
2  Point p2 = new Point(4, 6);
3  p1.moveUp(3.5);
4  p2.moveUp(4.7);
```

   ○ p1 and p2 are called the *call targets* or *context objects*.
   ○ **Lines 3 and 4** apply the same definition of the moveUp method.
   ○ But how does Java distinguish the change to p1.y versus the change to p2.y?

The `this` Reference (2)

- In the *method* definition, each *attribute* has an *implicit* `this` which refers to the <mark>context object</mark> in a call to that method.

```java
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    this.x = newX;
    this.y = newY;
  }
  public void moveUp(double units) {
    this.y = this.y + units;
  }
}
```

- Each time when the *class* definition is used to create a new `Point` *object*, the `this` reference is substituted by the name of the new object.

The `this` Reference (3)

- After we create p1 as an instance of Point

```
Point p1 = new Point(2, 3);
```

- When invoking p1.moveUp(3.5), a version of moveUp that is specific to p1 will be used:

```java
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    p1 .x = newX;
    p1 .y = newY;
  }
  public void moveUp(double units) {
    p1 .y = p1 .y + units;
  }
}
```

The `this` Reference (4)

- After we create p2 as an instance of Point

```
Point p2 = new Point(4, 6);
```

- When invoking p2.moveUp(4.7), a version of moveUp that is specific to p2 will be used:

```java
public class Point {
  private double x;
  private double y;
  public Point(double newX, double newY) {
    p2 .x = newX;
    p2 .y = newY;
  }
  public void moveUp(double units) {
    p2 .y = p2 .y + units;
  }
}
```

The `this` Reference (5)

The `this` reference can be used to **disambiguate** when the names of *input parameters* clash with the names of *class attributes*.

```java
public class Point {
  private double x;
  private double y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public void setX(double x) {
    this.x = x;
  }
  public void setY(double y) {
    this.y = y;
  }
}
```

The `this` Reference (6.1): Common Error

The following code fragment compiles but is problematic:

```
1  public class Person {
2    private String name;
3    private int age;
4    public Person(String name, int age) {
5      name = name;
6      age = age;
7    }
8    public void setAge(int age) {
9      age = age;
10   }
11 }
```

- Why?   [variable **shadowing**]
  Target (LHS) of the assignment (L5) refers to parameter `name` (L4).
- Fix?

Always remember to use `this` when *input parameter* names
clash with *class attribute* names.

```java
public class Person {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public void setAge(int age) {
    this.age = age;
  }
}
```

OOP: Mutator Methods

- These methods *change* values of attributes.
- We call such methods *mutators* (with `void` return type).

```
public class Person {
  ...
  public void gainWeight(double units) {
    this.weight = this.weight + units;
  }
}
```

```
public class Point {
  ...
  public void moveUp() {
    this.y = this.y + 1;
  }
}
```

OOP: Accessor Methods

- These methods *return* the result of computation based on attribute values.
- We call such methods **accessors** (with non-`void` return type).

```
public class Person {
 ...
 public double getBMI() {
   double bmi = this.height / (this.weight * this.weight);
   return bmi;
 } }
```

```
public class Point {
 ...
 public double getDistanceFromOrigin() {
   double dist =
       Math.sqrt(this.x * this.x + this.y * this.y);
   return dist;
 } }
```

OOP: Method Calls

```
1   Point p1 = new  Point (3, 4);
2   Point p2 = new  Point (-4, -3);
3   System.out.println(p1. getDistanceFromOrigin() );
4   System.out.println(p2. getDistanceFromOrigin() );
5   p1. moveUp(1) ;
6   p2. moveUp(1) ;
7   System.out.println(p1. getDistanceFromOrigin() );
8   System.out.println(p2. getDistanceFromOrigin() );
```

- **Lines 1 and 2** create two different instances of `Point`
- **Lines 3 and 4:** invoking the same accessor method on two different instances returns *distinct* values
- **Lines 5 and 6:** invoking the same mutator method on two different instances results in *independent* changes
- **Lines 3 and 7:** invoking the same accessor method on the same instance *may* return *distinct* values, why?   **Line 5**

See the lecture recording on tracing the above program *here*.

OOP: Use of Mutator vs. Accessor Methods

- Calls to **_mutator methods_** _cannot_ be used as values.
  - e.g., `System.out.println(jim.setWeight(78.5));` ✗
  - e.g., `double w = jim.setWeight(78.5);` ✗
  - e.g., `jim.setWeight(78.5);` ✓
- Calls to **_accessor methods_** _should_ be used as values.
  - e.g., `jim.getBMI();` ✗
  - e.g., `System.out.println(jim.getBMI());` ✓
  - e.g., `double w = jim.getBMI();` ✓

OOP: Method Parameters

- **Principle 1:** A `constructor` needs an *input parameter* for every attribute that you wish to initialize.

  e.g., `Person(double w, double h)` vs. `Person(String fName, String lName)`

- **Principle 2:** A `mutator` method needs an *input parameter* for every attribute that you wish to modify.

  e.g., In `Point`, `void moveToXAxis()` vs. `void moveUpBy(double unit)`

- **Principle 3:** An `accessor method` needs *input parameters* if the attributes alone are not sufficient for the intended computation to complete.

  e.g., In `Point`, `double getDistFromOrigin()` vs. `double getDistFrom(Point other)`

OOP: Reference Aliasing (1)

```
1  int i = 3;
2  int j = i;   System.out.println(i == j);/*true*/
3  int k = 3;   System.out.println(k == i && k == j);/*true*/
```

- **Line 2** copies the number stored in i to j.
- After **Line 4**, i, j, k refer to three separate integer placeholder, which happen to store the same value 3.

```
1  Point p1 = new Point(2, 3);
2  Point p2 = p1;   System.out.println(p1 == p2);/*true*/
3  Point p3 = new Point(2, 3);
4  Systme.out.println(p3 == p1 || p3 == p2);/*false*/
5  Systme.out.println(p3.x == p1.x && p3.y == p1.y);/*true*/
6  Systme.out.println(p3.x == p2.x && p3.y == p2.y);/*true*/
```

- **Line 2** copies the *address* stored in p1 to p2.
- Both p1 and p2 refer to the same object in memory!
- p3, whose *contents* are same as p1 and p2, refer to a different object in memory.

OOP: Reference Aliasing (2.1)

Problem: Consider assignments to *primitive* variables:

```
1   int i1 = 1;
2   int i2 = 2;
3   int i3 = 3;
4   int[] numbers1 = {i1, i2, i3};
5   int[] numbers2 = new int[numbers1.length];
6   for(int i = 0; i < numbers1.length; i ++) {
7     numbers2[i] = numbers1[i];
8   }
9   numbers1[0] = 4;
10  System.out.println(numbers1[0]);
11  System.out.println(numbers2[0]);
```

OOP: Reference Aliasing (2.2)

Exercise: Consider assignments to *reference* variables:

```
1  Person alan = new Person("Alan");
2  Person mark = new Person("Mark");
3  Person tom = new Person("Tom");
4  Person jim = new Person("Jim");
5  Person[] persons1 = {alan, mark, tom};
6  Person[] persons2 = new Person[persons1.length];
7  for(int i = 0; i < persons1.length; i ++) {
8    persons2[i] = persons1[i]; }
9  persons1[0].setAge(70);
10 System.out.println(jim.getAge());
11 System.out.println(alan.getAge());
12 System.out.println(persons2[0].getAge());
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.getAge());
16 System.out.println(alan.getAge());
17 System.out.println(persons2[0].getAge());
```

See the lecture recording on tracing the above program *here*.

Java Data Types (3.1)

- An *attribute* may store the reference to another object.

```
public class Person { private Person spouse; }
```

- Methods may take as *parameters* references to other objects.

```
public class Person {
  public void marry(Person other) { ... } }
```

- *Return values* from methods may be references to objects.

```
public class Point {
  public void moveUpBy(int i) { y = y + i; }
  Point movedUpBy(int i) {
    Point np = new Point(x, y);
    np.moveUpBy(i);
    return np;
  }
}
```

See the lecture recording on tracing the above program *here*.

Java Data Types (3.2.1)

An attribute may be **_multi_**-valued, **_reference_**-typed
e.g., of type `Point[]`, storing references to Point objects.

```
1  public class PointCollector {
2    private Point[] points; private int nop;/* number of points */
3    public PointCollector() { this.points = new Point[100]; }
4    public void addPoint(double x, double y) {
5      this.points[this.nop] = new Point(x, y); this.nop++; }
6    public Point[] getPointsInQuadrantI() {
7      Point[] ps = new Point[this.nop];
8      int count = 0; /* number of points in Quadrant I */
9      for(int i = 0; i < this.nop; i++) {
10       Point p = this.points[i];
11       if(p.x > 0 && p.y > 0) { ps[count] = p; count++; } }
12     Point[] q1Points = new Point[count];
13     /* ps contains null if count < nop */
14     for(int i = 0; i < count; i++) { q1Points[i] = ps[i] }
15     return q1Points;
16   } }
```

Required Reading: Point and PointCollector

```
1   public class PointCollectorTester {
2     public static void main(String[] args) {
3       PointCollector pc = new PointCollector();
4       System.out.println(pc.getNumberOfPoints()); /* 0 */
5       pc.addPoint(3, 4);
6       System.out.println(pc.getNumberOfPoints()); /* 1 */
7       pc.addPoint(-3, 4);
8       System.out.println(pc.getNumberOfPoints()); /* 2 */
9       pc.addPoint(-3, -4);
10      System.out.println(pc.getNumberOfPoints()); /* 3 */
11      pc.addPoint(3, -4);
12      System.out.println(pc.getNumberOfPoints()); /* 4 */
13      Point[] ps = pc.getPointsInQuadrantI();
14      System.out.println(ps.length);  /* 1 */
15      System.out.println("(" +
16        ps[0].getX() + ", " + ps[0].getY() + ")"); /* (3, 4) */
17    }
18  }
```

See the lecture recording on tracing the above program *here*.

Anonymous Objects (1)

- What's the difference between these two fragments of code?

```
1  double square(double x) {
2    double sqr = x * x;
3    return sqr; }
```

```
1  double square(double x) {
2    return x * x; }
```

After **L2**, the result of $x * x$:
- ○ LHS: it can be <u>reused</u> (without recalculating) via the name sqr.
- ○ RHS: it is <u>not stored</u> anywhere and returned right away.

- Same principles applies to objects:

```
1  Person getP(String n) {
2    Person p = new Person(n);
3    return p; }
```

```
1  Person getP(String n) {
2    return new Person(n); }
```

new Person(n) is an object whose address is not stored in a variable.
- ○ LHS: **L2** stores the address of this anonymous object in p.
- ○ RHS: **L2** returns the address of this anonymous object directly.

Anonymous Objects (2.1)

Anonymous objects can also be used as *assignment sources* or *argument values*:

```
class Member {
 private Order[] orders;
 private int noo;
 /* constructor ommitted */
 public void addOrder(Order o) {
   this.orders[this.noo] = o;
   this.noo++;
 }
 public void addOrder(String n, double p, double q) {
   this.addOrder( new Order(n, p, q) );
   /* Equivalent implementation:
    * this.orders[this.noo] = new Order(n, p, q); noo ++;
    */
 }
}
```

Anonymous Objects (2.2)

One more example on using anonymous objects:

```java
public class MemberTester {
 public static void main(String[] args) {
   Member m = new Member("Alan");
   Order o = new Order("Americano", 4.7, 3);
   m.addOrder(o);
   m.addOrder( new Order("Cafe Latte", 5.1, 4) );
  }
}
```

Consider the `Person` class

```
public class Person {
  private String name;
  private Person spouse;
  public Person(String name) {
    this.name = name;
  }
}
```

How do you implement a mutator method `marry` which marries the current `Person` object to an input `Person` object?

```java
public void marry(Person other) {
  if(this.spouse != null || other.spouse != null) {
    /* Error: both must be single */
  }
  else { this.spouse = other; other.spouse = this; }
}
```

When we call jim.marry(elsa): this is substituted by the *context object* jim, and other by the *argument* elsa.

```java
public void marry(Person ~~other~~ elsa) {
  ...
    jim.spouse = elsa;
    elsa.spouse = jim;
  ...
}
```

OOP: The Dot Notation (2)

- **LHS** of dot *can be more complicated than a variable*:

  ○ It can be a *path* that brings you to an object

  ```java
  public class Person {
    private String name;/* public accessor: name() */
    private Person spouse;/* public accessor: spouse() */
  }
  ```

  ○ Say we have `Person jim = new Person("Jim Davies")`
  ○ Inquire about `jim`'s name?   [`jim.name()`]
  ○ Inquire about `jim`'s spouse's name?   [`jim.spouse().name()`]
  ○ But what if `jim` is single (i.e., `jim.spouse() == null`)?
    Calling `jim.spouse().name()` will cause *NullPointerException*!!
  ○ **Quesion.** Assuming that:
    • `jim` is not single.   [`jim.spouse() != null`]
    • The marriage is mutual.   [`jim.spouse().spouse() != null`]
    What does `jim.spouse().spouse().name()` mean?

    Answer. `jim.name()`

OOP: Helper Methods (1)

- <u>After</u> you complete and test your program, feeling confident that it is *correct*, you may find that there are lots of *repetitions*.
- When similar fragments of code appear in your program, we say that your code "*smells*"!
- We may eliminate *repetitions* of your code by:
  - *Factoring out* recurring code fragments into a new method.
  - This new method is called a *helper method* :
    - You can replace <u>every occurrence</u> of the recurring code fragment by a *call* to this helper method, with appropriate argument values.
    - That is, we *reuse* the body implementation, rather than repeating it over and over again, of this helper method via calls to it.
- This process is called *refactoring* of your code:
  <u>Modify the code structure</u> **without** compromising *correctness*.

  See the lecture recording on helper methods *here*.

OOP: Helper (Accessor) Methods (2.1)

```
public class PersonCollector {
 private Person[] ps;
 private final int MAX = 100;/* max # of persons to store */
 private int nop; /* number of persons */
 public PersonCollector() {
   this.ps = new Person[MAX];
 }
 public void addPerson(Person p) {
   this.ps[this.nop] = p;
   this.nop++;
 }
 /* Tasks:
  * 1. An accessor: boolean personExists(String n)
  * 2. A mutator: void changeWeightOf(String n, double w)
  * 3. A mutator: void changeHeightOf(String n, double h)
  */
}
```

OOP: Helper (Accessor) Methods (2.2.1)

```
public class PersonCollector {
 /* ps, MAX, nop, PersonCollector(), addPerson */
 public boolean personExists(String n) {
   boolean found = false;
   for(int i = 0; i < nop; i ++) {
     if(ps[i].getName().equals(n)) { found = true; } }
   return found;
 }
 public void changeWeightOf(String n, double w) {
   for(int i = 0; i < nop; i ++) {
     if(ps[i].getName().equals(n)) { ps[i].setWeight(w); } }
 }
 public void changeHeightOf(String n, double h) {
   for(int i = 0; i < nop; i ++) {
     if(ps[i].getName().equals(n)) { ps[i].setHeight(h); } }
 }
}
```

OOP: Helper (Accessor) Methods (2.2.2)

```java
public class PersonCollector {/* code smells:repetitions! */
 /* ps, MAX, nop, PersonCollector(), addPerson */
 public boolean personExists( String n ) {
   boolean found = false;
   for(int i = 0; i < nop; i ++) {
     if(ps[i].getName().equals(n)) { found = true; } }
   return found;
 }
 public void changeWeightOf( String n , double w) {
   for(int i = 0; i < nop; i ++) {
     if(ps[i].getName().equals(n)) { ps[i] .setWeight(w); } }
 }
 public void changeHeightOf( String n , double h) {
   for(int i = 0; i < nop; i ++) {
     if(ps[i].getName().equals(n)) { ps[i] .setHeight(h); } }
 }
}
```

```
public class PersonCollector { /* Code Smell Eliminated */
 /* ps, MAX, nop, PersonCollector(), addPerson */
 private int indexOf (String n) { /* Helper Methods */
   int i = -1;
   for(int j = 0; j < nop; j ++) {
     if(ps[j].getName().equals(n)) { i = j; }
   }
   return i; /* -1 if not found; >= 0 if found. */
 }
 public boolean personExists(String n) {
   return this.indexOf (n) >= 0; }
 public void changeWeightOf(String n, double w) {
   int i = indexOf (n); if(i >= 0) { ps[i].setWeight(w); }
 }
 public void changeHeightOf(String n, double h) {
   int i = indexOf (n); if(i >= 0) { ps[i].setHeight(h); }
 }
}
```

OOP: Helper (Accessor) Methods (3.1)

Problems:

- A `Point` class with `x` and `y` coordinate values.

- Accessor double *getDistanceFromOrigin*().

  p.getDistanceFromOrigin() returns the distance between p and (0, 0).

- Accessor double *getDistancesTo*(Point p1, Point p2).

  p.getDistancesTo(p1, p2) returns the sum of distances between p and p1, and between p and p2.

- Accessor double *getTriDistances*(Point p1, Point p2).

  p.getDistancesTo(p1, p2) returns the sum of distances between p and p1, between p and p2, and between p1 and p2.

OOP: Helper (Accessor) Methods (3.2)

```
class Point { /* code smells:repetitions! */
 double x; double y;

 double getDistanceFromOrigin() {
  return Math.sqrt(Math.pow(this.x - 0, 2) + Math.pow(this.y - 0, 2));

 double getDistancesTo(Point p1, Point p2) {
  return
   Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
   +
   Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2)); }

 double getTriDistances(Point p1, Point p2) {
  return
   Math.sqrt(Math.pow(this.x - p1.x, 2) + Math.pow(y - p1.y, 2))
   +
   Math.sqrt(Math.pow(this.x - p2.x, 2) + Math.pow(y - p2.y, 2))
   +
   Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
 }
}
```

OOP: Helper (Accessor) Methods (3.3)

- The code pattern

  ```
  Math.sqrt(Math.pow(... - ..., 2) + Math.pow(... - ..., 2))
  ```

  is written down explicitly every time we need to use it.

- Create a *helper method* out of it, with the right *parameter* and *return* types:

```
double getDistanceFrom(double otherX, double otherY) {
   return Math.sqrt(
     Math.pow(ohterX - this.x, 2)
     +
     Math.pow(otherY - this.y, 2));
}
```

OOP: Helper (Accessor) Methods (3.4)

```
public class Point { /* Code Smell Eliminated */
 private double x; private double y;
 double getDistanceFrom(double otherX, double otherY) {
  return Math.sqrt(Math.pow(ohterX - this.x, 2) +
       Math.pow(otherY - this.y, 2));
 }
 double getDistanceFromOrigin() {
  return this.getDistanceFrom(0, 0);
 }
 double getDistancesTo(Point p1, Point p2) {
  return this.getDistanceFrom(p1.x, p1.y) +
       this.getDistanceFrom(p2.x, p2.y);
 }
 double getTriDistances(Point p1, Point p2) {
  return this.getDistanceFrom(p1.x, p1.y) +
       this.getDistanceFrom(p2.x, p2.y) +
       p1.getDistanceFrom(p2.x, p2.y)
 }
}
```

OOP: Helper (Mutator) Methods (4.1)

```java
public class Student {
 private String name;
 private double balance;
 public Student(String n, double b) {
   name = n;
   balance = b;
 }

 /* Tasks:
  * 1. A mutator void receiveScholarship(double val)
  * 2. A mutator void payLibraryOverdue(double val)
  * 3. A mutator void payCafeCoupons(double val)
  * 4. A mutator void transfer(Student other, double val)
  */
}
```

OOP: Helper (Mutator) Methods (4.2.1)

```
public class Student {
 /* name, balance, Student(String n, double b) */
 public void receiveScholarship(double val) {
  balance = balance + val;
 }
 public void payLibraryOverdue(double val) {
  balance = balance - val;
 }
 public void payCafeCoupons(double val) {
  balance = balance - val;
 }
 public void transfer(Student other, double val) {
  balance = balance - val;
  other.balance = other.balance + val;
 }
}
```

```
public class Student { /* code smells:repetitions! */
 /* name, balance, Student(String n, double b) */
 public void receiveScholarship(double val) {
   balance = balance + val;
 }
 public void payLibraryOverdue(double val) {
   balance = balance - val;
 }
 public void payCafeCoupons(double val) {
   balance = balance - val;
 }
 public void transfer(Student other, double val) {
   balance = balance - val;
   balance = other.balance + val;
 }
}
```

OOP: Helper (Mutator) Methods (4.3)

```
public class Student { /* Code Smell Eliminated */
 /* name, balance, Student(String n, double b) */
 public void deposit (double val) { /* Helper Method */
  balance = balance + val;
 }
 public void withdraw (double val) { /* Helper Method */
  balance = balance - val;
 }
 public void receiveScholarship(double val) { this. deposit (val); }
 public void payLibraryOverdue(double val) { this. withdraw (val); }
 public void payCafeCoupons(double val) { this. withdraw (val) }
 public void transfer(Student other, double val) {
  this. withdraw (val);
  other. deposit (val);
 }
}
```

Static Variables (1)

```java
public class Account {
  private int id;
  private String owner;
  public int getID() { return this.id; }
  public Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

```java
class AccountTester {
  Account acc1 = new Account(1, "Jim");
  Account acc2 = new Account(2, "Jeremy");
  System.out.println(acc1.getID() != acc2.getID());
}
```

But, managing the unique id's *manually* is *error-prone* !

Static Variables (2)

```
class Account {
  private static int globalCounter = 1 ;
  private int id; String owner;
  public Account(String owner) {
    this.id = globalCounter ;
    globalCounter ++ ;
    this.owner = owner; } }
```

```
class AccountTester {
  Account acc1 = new Account("Jim");
  Account acc2 = new Account("Jeremy");
  System.out.println(acc1.getID() != acc2.getID()); }
```

- Each instance of a class (e.g., acc1, acc2) has a *local* copy of each attribute or instance variable (e.g., id).
  - Changing acc1.id does not affect acc2.id.
- A *static* variable (e.g., globalCounter) belongs to the class.
  - All instances of the class <u>share</u> a *single* copy of the *static* variable.
  - Change to globalCounter via acc1 is also visible to acc2.

Static Variables (3)

```
public class Account {
  private static int globalCounter = 1;
  private int id; private String owner;
  public Account(String owner) {
    this.id = globalCounter;
    globalCounter ++;
    this.owner = owner;
  } }
```

- *Static* variable globalCounter is not instance-specific like *instance* variable (i.e., attribute) id is.
- To access a *static* variable:
  - **No** context object is needed.
  - Use of the class name suffices, e.g., Account.globalCounter.
- Each time Account's constructor is called to create a new instance, the increment effect is *visible to all existing objects* of Account.

Static Variables (4.1): Common Error

```
public class Client {
  private Account[] accounts;
  private  static  int numberOfAccounts = 0;
  public void addAccount(Account acc) {
    accounts[this.numberOfAccounts] = acc;
    this.numberOfAccounts ++;
  } }
```

```
public class ClientTester {
  Client bill = new Client("Bill");
  Client steve = new Client("Steve");
  Account acc1 = new Account();
  Account acc2 = new Account();
  bill.addAccount(acc1);
    /* correctly added to bill.getAccounts()[0] */
  steve.addAccount(acc2);
    /* mistakenly added to steve.getAccounts()[1]! */
}
```

Static Variables (4.2): Common Error

- Attribute numberOfAccounts should **not** be declared as static as its value should be specific to the client object.
- If it were declared as static, then every time the addAccount method is called, although on different objects, the increment effect of numberOfAccounts will be visible to all Client objects.
- Here is the correct version:

```java
public class Client {
  private Account[] accounts;
  private int numberOfAccounts;
  public void addAccount(Account acc) {
    accounts[this.numberOfAccounts] = acc;
    this.numberOfAccounts ++;
  }
}
```

Static Variables (5.1): Common Error

```
1  public class Bank {
2     private string branchName;
3     public String getBrachName() { return this.branchName; }
4     private static int nextAccountNumber = 0;
5     public static String getInfo() {
6        nextAccountNumber++;
7        return this.branchName + nextAccountNumber;
8     }
9  }
```

- *Non-static method cannot be referenced from a static context*
- **Line 4** declares that we `can` call the method `getInfo` without instantiating an object of the class `Bank`.
- However, in **Line 7**, the *static* method references a *non-static* attribute, for which we `must` instantiate a `Bank` object.

Static Variables (5.2): Common Error

```
1  public class Bank {
2     private String branchName;
3     public String getBrachName() { return this.branchName; }
4     private static int nextAccountNumber = 0;
5     public static String getInfo() {
6        nextAccountNumber++;
7        return this.branchName + nextAccountNumber;
8     }
9  }
```

- To call `getInfo()`, no instances of `Bank` are required:

```
Bank .getInfo();
```

- *Contradictorily*, to access `branchName`, a *context object* is required:

```
Bank b = new Bank(); b.setBranch("Songdo IBK");
System.out.println( b .getBranchName());
```

There are two possible ways to fix:

1. Remove all uses of *non-static* variables (i.e., `branchName`) in the *static* method (i.e., `useAccountNumber`).

2. Declare `branchName` as a *static* variable.
   - This does not make sense.
     ∵ `branchName` should be a value specific to each `Bank` instance.

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (7)

Exceptions

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

- <u>Caller</u> vs. <u>Callee</u> in a Method Invocation
- ***Error Handling*** via <u>Console</u> Message
- The ***Catch***-or-***Specify*** Requirement
- Example: To Handle or <u>Not</u> to Handle?
- ***Error Handling*** via <u>Exceptions</u>
- What to Do When an <u>Exception</u> is Thrown at <u>Runtime</u>
- More Examples on <u>Exception Handling</u>

Caller vs. Callee

- Within the body implementation of a method ({ . . . }), we may call other methods.

```
1  class C1 {
2    void m1() {
3      C2 o = new C2();
4      o.m2(); /* static type of o is C2 */
5    }
6  }
```

- From **Line 4**, we say:
  - Method *C1.m1* (i.e., method m1 from class C1) is the *caller* of method *C2.m2*.
  - Method *C2.m2* is the *callee* of method *C1.m1*.

Stack of Method Calls

- Execution of a Java project *starts* from the `main` **method** of some class (e.g., CircleTester, BankApplication).
- Each line of *method call* involves the execution of that method's *body implementation*
  - That method's body implementation may also involve *method calls*, which may in turn involve more *method calls*, and *etc*.
  - It is typical that we end up with *a chain of method calls* !
  - We visualize this chain of method calls as a *call stack* . For example:
    - Account.withdraw   [*top* of stack; *latest* called]
    - Bank.withdrawFrom
    - BankApplication.main   [*bottom* of stack; *earliest* called]
  - The closer a method is to the *top* of the call stack, the *later* its call was made.

Error Reporting via Consoles: Circles (1)

```
1  class Circle {
2    double radius;
3    Circle() { /* radius defaults to 0 */ }
4    void setRadius(double r) {
5      if ( r < 0 ) { System.out.println( "Invalid radius." ); }
6      else { radius = r; }
7    }
8    double getArea() { return radius * radius * 3.14; }
9  }
```

- A negative radius is considered as an *invalid input value* to method setRadius.
- What if the *caller* of Circle.setRadius passes a negative value for r?
  - An error message is *printed to the console* (**Line 5**) to warn the *caller* of setRadius.
  - However, printing an error message to the console *does not force* the *caller* of setRadius to stop and handle invalid values of r.

```
1   class CircleCalculator {
2    public static void main(String[] args) {
3      Circle c = new Circle();
4      c.setRadius( -10 );
5      double area = c.getArea();
6      System.out.println("Area: " + area);
7    }
8   }
```

- **L4**: *CircleCalculator.main* is *caller* of *Circle.setRadius*
- A negative radius is passed to setRadius in **Line 4**.
- The execution *always flows smoothly* from **Lines 4** to **Line 5**, *even when there was an error* message printed from **Line 4**.
- It is not feasible to check if there is any kind of error message printed to the console right after the execution of **Line 4**.
- **Solution:** A way to underline force *CircleCalculator.main*, *caller* of *Circle.setRadius*, to realize that things might go wrong.
  ⇒ When things do go wrong, <u>immediate</u> actions are needed.

Error Reporting via Consoles: Bank (1)

```java
class Account {
  int id; double balance;
  Account(int id) { this.id = id; /* balance defaults to 0 */ }
  void deposit(double a) {
    if ( a < 0 ) { System.out.println( "Invalid deposit." ); }
    else { balance += a; }
  }
  void withdraw(double a) {
    if ( a < 0 || balance - a < 0 ) {
      System.out.println( "Invalid withdraw." ); }
    else { balance -= a; }
  }
}
```

- A negative deposit or withdraw amount is *invalid*.
- When an *error* occurs, a message is *printed to the console*.
- However, printing error messages does not <u>force</u> the *caller* of
  Account.deposit or Account.withdraw to stop and
  handle invalid values of a.

Error Reporting via Consoles: Bank (2)

```
1   class Bank {
2     Account[] accounts; int numberOfAccounts;
3     Account(int id) { ... }
4     void withdrawFrom(int id, double a) {
5       for(int i = 0; i < numberOfAccounts; i ++) {
6         if(accounts[i].id == id) {
7           accounts[i].withdraw( a );
8         }
9       } /* end for */
10    } /* end withdraw */
11  }
```

- **L7**: *Bank.withdrawFrom* is *caller* of *Account.withdraw*
- What if in **Line 7** the value of a is negative?
  Error message ⟨Invalid withdraw⟩ printed from method
  Account.withdraw to console.
- Impossible to <u>force</u> *Bank.withdrawFrom*, the *caller* of
  Account.withdraw, to stop and handle invalid values of a.

Error Reporting via Consoles: Bank (3)

```
1    class BankApplication {
2      pubic static void main(String[] args) {
3        Scanner input = new Scanner(System.in);
4        Bank b = new Bank(); Account acc1 = new Account(23);
5        b.addAccount(acc1);
6        double a = input.nextDouble();
7        b.withdrawFrom(23, a );
8        System.out.println("Transaction Completed.");
9      }
```

- There is a chain of method calls:
  - *BankApplication.main* calls **Bank.withdrawFrom**
  - **Bank.withdrawFrom** calls *Account.withdraw*.
- The actual update of balance occurs at the Account class.
  - What if in **Line 7** the value of a is negative?
    Invalid withdraw printed from *Bank.withdrawFrom*,
    originated from *Account.withdraw* to console.
  - However, impossible to <u>stop</u> *BankApplication.main* from
    continuing to execute **Line 8**, printing Transaction Completed.
- **Solution:** Define error checking only once and let it *propagate*.

What is an Exception?

- An **exception** is an *event*, which
  - occurs during the *execution of a program*
  - *disrupts the normal flow* of the program's instructions
- When an error occurs within a method:
  - the method throws an exception:
    - first creates an *exception object*
    - then hands it over to the *runtime system*
  - the exception object contains information about the error:
    - type   [e.g., `NegativeRadiusException`]
    - the state of the program when the error occurred

What to Do When an Exception Is Thrown? (LASSONDE)

- After a method *throws an exception*, the *runtime system* searches the corresponding <mark>call stack</mark> for a method that contains a block of code to *handle* the exception.
  - This block of code is called an <mark>*exception handler*</mark>.
    - An exception handler is **appropriate** if the *type* of the *exception object thrown* matches the *type* that can be handled by the handler.
    - The exception handler chosen is said to *catch* the exception.
  - The search goes from the *top* to the *bottom* of the call stack:
    - The method in which the *error* occurred is searched first.
    - The *exception handler* is not found in the current method being searched ⇒ Search the method that calls the current method, and *etc*.
    - When an appropriate *handler* is found, the *runtime system* passes the exception to the handler.
  - The *runtime system* searches all the methods on the <mark>call stack</mark> without finding an **appropriate** *exception handler* ⇒ The program terminates and the exception object is directly "thrown" to the console!
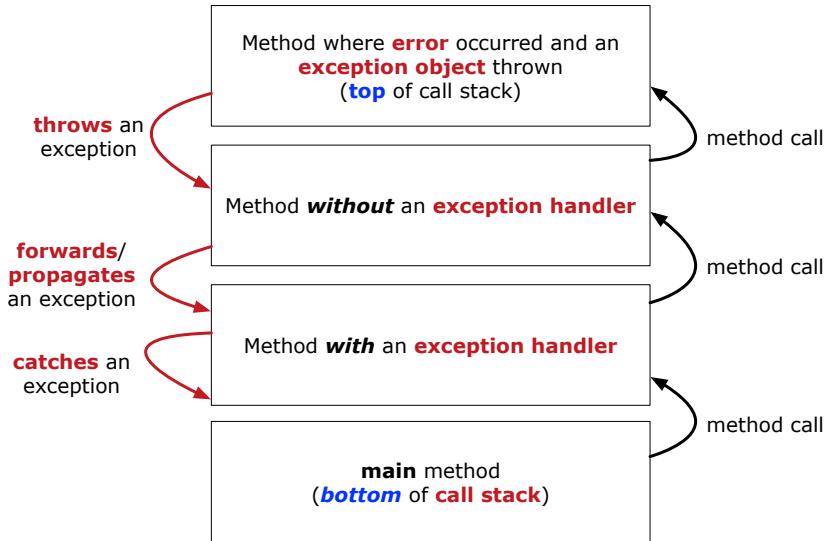
Code (e.g., a method call) that might throw certain exceptions
must be enclosed by one of the two ways:

1. The "Catch" Solution: A `try` statement that *catches* and
handles the *exception*

(**without** propagating that exception to the method's *caller*).

```
main(...) {
  Circle c = new Circle();
  try {
    c.setRadius(-10);
  }
  catch(NegativeRaidusException e) {
    ...
  }
}
```

Code (e.g., a method call) that might throw certain exceptions must be enclosed by one of the two ways:

2. **The "Specify" Solution:** A method that specifies as part of its *header* that it may (or may not) ***throw*** the ***exception*** (which will be thrown to the method's ***caller*** for handling).

```
class Bank {
  Account[] accounts; /* attribute */
  void withdraw (double amount)
    throws InvalidTransactionException {
    ...
    accounts[i].withdraw(amount);
    ...
  }
}
```

Consider the following three classes:

```java
class A {
  ma(int i) {
    if(i < 0) { /* Error */ }
    else { /* Do something. */ }
  } }
```

```java
class B {
  mb(int i) {
    A oa = new A();
    oa.ma(i); /* Error occurs if i < 0 */
  } }
```
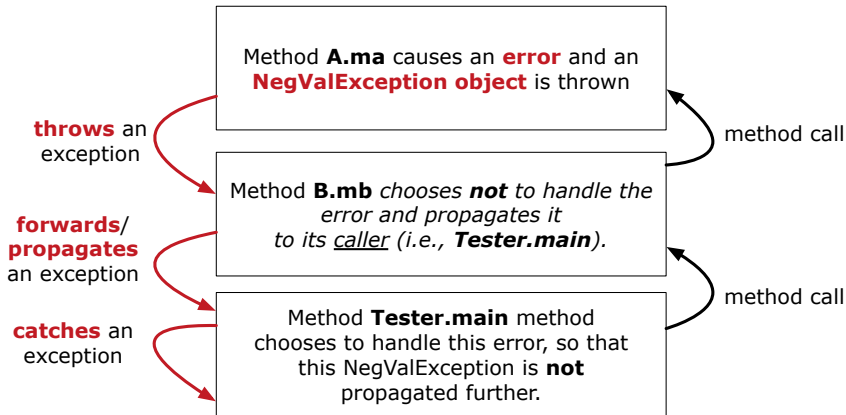
```java
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Where can the error be handled? */
  } }
```

- We assume the following kind of error for negative values:

```
class NegValException extends Exception {
  NegValException(String s) { super(s); }
}
```

- The above kind of exception may be thrown by calling `A.ma`.
- We will see three kinds of possibilities of handling this exception:

  Version 1:
  Handle it in `B.mb`
  Version 2:
  Pass it from `B.mb` and handle it in `Tester.main`
  Version 3:
  Pass it from `B.mb`, then from `Tester.main`, then throw it to the console.

Version 1: Handle the exception in `B.mb`.
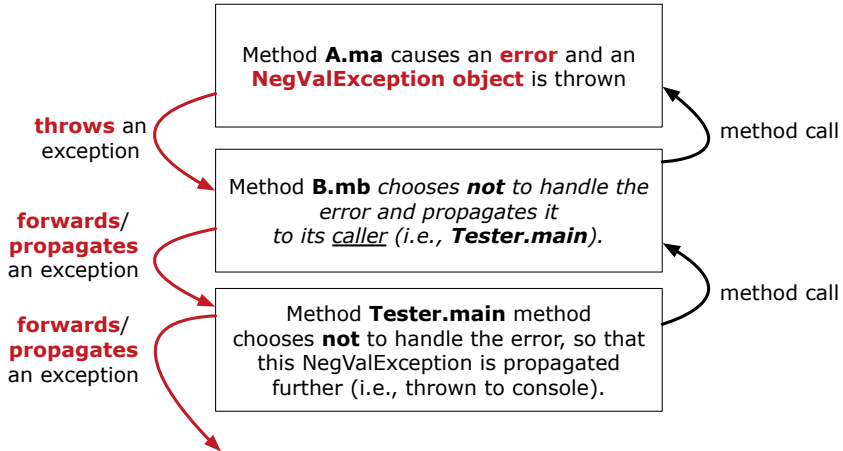
```java
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```java
class B {
  mb(int i) {
    A oa = new A();
    try { oa.ma(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

```java
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i); /* Error, if any, would have been handled in B.mb. */
  } }
```

Version 1: Handle the exception in `B.mb`.

Method **A.ma** causes an **error** and an **NegValException object** is thrown

throws an exception

method call

Method **B.mb** *chooses to handle the error right away using a **try-catch** block.*

catches an exception

method call

Method **Tester.main** method need not worry about this error.
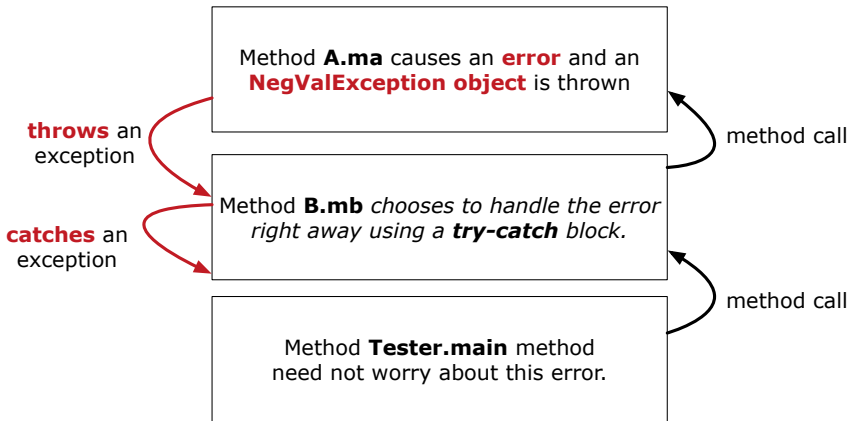
Example: to Handle or Not to Handle? (3.1)

Version 2: Handle the exception in `Tester.main`.

```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
  } }
```

```
class Tester {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    try { ob.mb(i); }
    catch(NegValException nve) { /* Do something. */ }
  } }
```

Version 2: Handle the exception in `Tester.main`.

throws an exception

Method **A.ma** causes an **error** and an **NegValException object** is thrown

method call

forwards/ **propagates** an exception

Method **B.mb** *chooses **not** to handle the error and propagates it to its* <u>caller</u> *(i.e., **Tester.main**).*

method call

catches an exception

Method **Tester.main** method chooses to handle this error, so that this NegValException is **not** propagated further.

Example: to Handle or Not to Handle? (4.1)

Version 3: Handle in neither of the classes.

```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
class B {
  mb(int i) throws NegValException {
    A oa = new A();
    oa.ma(i);
  } }
```

```
class Tester {
  public static void main(String[] args) throws NegValException {
    Scanner input = new Scanner(System.in);
    int i = input.nextInt();
    B ob = new B();
    ob.mb(i);
  } }
```

Version 3: Handle in neither of the classes.

Method **A.ma** causes an **error** and an **NegValException object** is thrown

throws an exception

method call

Method **B.mb** *chooses **not** to handle the error and propagates it to its <u>caller</u> (i.e., **Tester.main**).*

forwards/ **propagates** an exception

method call

Method **Tester.main** method chooses **not** to handle the error, so that this NegValException is propagated further (i.e., thrown to console).

forwards/ **propagates** an exception

LASSONDE
SCHOOL OF ENGINEERING

```java
public class InvalidRadiusException extends Exception {
 public InvalidRadiusException(String s) {
   super(s);
 }
}
```

- A new kind of Exception: InvalidRadiusException
- For any method that can have this kind of error, we declare at that method's *header* that it may *throw* an InvalidRaidusException object.

```
class Circle {
 double radius;
 Circle() { /* radius defaults to 0 */ }
 void setRadius(double r) throws InvalidRadiusException {
   if (r < 0) {
     throw new InvalidRadiusException("Negative radius.");
   }
   else { radius = r; }
 }
 double getArea() { return radius * radius * 3.14; }
}
```

- As part of the *header* of setRadius, we declare that it may
  throw an InvalidRadiusException object at runtime.
- Any method that calls setRadius will be forced to
  deal with this potential error .

Error Reporting via Exceptions: Circles (3)

```
1   class CircleCalculator1 {
2    public static void main(String[] args) {
3      Circle c = new Circle();
4      try {
5        c.setRadius(-10);
6        double area = c.getArea();
7        System.out.println("Area: " + area);
8      }
9      catch(InvalidRadiusException e) {
10       System.out.println(e);
11     }
12   } }
```

- **Lines 6** is forced to be wrapped within a *try-catch* block, since it may *throw* an InvalidRadiusException object.

- If an InvalidRadiusException object is thrown from **Line 6**, then the normal flow of execution is *interrupted* and we go to the catch block starting from **Line 9**.

Exercise: Extend `CircleCalculator1`: repeatedly prompt for a new radius value until a valid one is entered (i.e., the `InvalidRadiusException` does not occur).

```
Enter a radius:
-5
Radius -5.0 is invalid, try again!
Enter a radius:
-1
Radius -1.0 is invalid, try again!
Enter a radius:
5
Circle with radius 5.0 has area: 78.5
```

```
1  public class CircleCalculator2 {
2    public static void main(String[] args) {
3      Scanner input = new Scanner(System.in);
4      boolean inputRadiusIsValid = false;
5      while (!inputRadiusIsValid) {
6        System.out.println("Enter a radius:");
7        double r = input.nextDouble();
8        Circle c = new Circle();
9        try { c.setRadius(r);
10            inputRadiusIsValid = true;
11           System.out.print("Circle with radius " + r);
12           System.out.println(" has area: "+ c.getArea()); }
13       catch(InvalidRadiusException e) { print("Try again!"); }
14     } } }
```

- At **L7**, if the user's input value is:
  - Non-Negative: **L8** – **L12**.　　　[inputRadiusIsValid set *true*]
  - Negative: **L8**, **L9**, **L13**.　　[inputRadiusIsValid remains *false*]

```java
public class InvalidTransactionException extends Exception {
 public InvalidTransactionException(String s) {
   super(s);
 }
}
```

- A new kind of `Exception`:
  `InvalidTransactionException`
- For any method that can have this kind of error, we declare at
  that method's *header* that it may *throw* an
  `InvalidTransactionException` object.

Error Reporting via Exceptions: Bank (2)

```
class Account {
 int id; double balance;
 Account() { /* balance defaults to 0 */ }
 void withdraw(double a) throws InvalidTransactionException {
   if (a < 0 || balance - a < 0) {
     throw new InvalidTransactionException("Invalid withdraw."); }
   else { balance -= a; }
 }
}
```

- As part of the *header* of `withdraw`, we declare that it may
  throw an `InvalidTransactionException` object at
  runtime.
- Any method that calls `withdraw` will be forced to
  deal with this potential error .

Error Reporting via Exceptions: Bank (3)

```
class Bank {
 Account[] accounts; int numberOfAccounts;
 Account(int id) { ... }
 void withdraw(int id, double a)
     throws InvalidTransactionException {
  for(int i = 0; i < numberOfAccounts; i ++) {
    if(accounts[i].id == id) {
      accounts[i].withdraw(a);
    }
  } /* end for */ } /* end withdraw */ }
```

- As part of the *header* of withdraw, we declare that it may *throw* an InvalidTransactionException object.

- Any method that calls withdraw will be forced to *deal with this potential error*.

- We are *propagating* the potential error for the right party (i.e., BankApplication) to handle.

```
1   class BankApplication {
2    pubic static void main(String[] args) {
3      Bank b = new Bank();
4      Account acc1 = new Account(23);
5      b.addAccount(acc1);
6      Scanner input = new Scanner(System.in);
7      double a = input.nextDouble();
8      try {
9        b.withdraw(23, a);
10       System.out.println(acc1.balance); }
11     catch (InvalidTransactionException e) {
12       System.out.println(e); } } }
```

- **Lines 9** is forced to be wrapped within a *try-catch* block, since it may *throw* an InvalidTransactionException object.

- If an InvalidTransactionException object is thrown from **Line 9**, then the normal flow of execution is interrupted and we go to the catch block starting from **Line 11**.

More Examples (1)

```
double r = ...;
double a = ...;
try{
  Bank b = new Bank();
  b.addAccount(new Account(34));
  b.deposit(34, 100);
  b.withdraw(34, a);
  Circle c = new Circle();
  c.setRadius(r);
  System.out.println(r.getArea());
}
catch(NegativeRadiusException e) {
  System.out.println(r + " is not a valid radius value.");
  e.printStackTrace();
}
catch(InvalidTransactionException e) {
  System.out.println(r + " is not a valid transaction value.");
  e.printStackTrace();
}
```

More Example (2.1)

The `Integer` class supports a method for parsing Strings:

```
public static int parseInt(String s)
                throws NumberFormatException
```

e.g., `Integer.parseInt("23")` returns 23

e.g., `Integer.parseInt("twenty-three")` throws a
`NumberFormatException`

Write a fragment of code that prompts the user to enter a string
(using `nextLine` from `Scanner`) that represents an integer.

If the user input is not a valid integer, then prompt them to enter
again.

More Example (2.2)

```java
Scanner input = new Scanner(System.in);
boolean validInteger = false;
while (!validInteger) {
  System.out.println("Enter an integer:");
  String userInput = input.nextLine();
  try {
    int userInteger = Integer.parseInt(userInput);
    validInteger = true;
  }
  catch(NumberFormatException e) {
    System.out.println(userInput + " is not a valid integer.");
    /* validInteger remains false */
  }
}
```

Beyond this lecture...

- Practice creating a new *exception* class upon a method throwing it in the body of implementation (e.g., `InvalidRadiusException`, `InvalidTransactionException`).
- Play with the source code:
  - `ExceptionsCircleAndBank.zip`
  - `ExceptionsToHandleOrNotToHandle.zip`

  Tip. Change input values so as to explore, in Eclipse *debugger*, possible (*normal* vs. *abnormal*) *execution paths* .

Index (1)

Index (2)

Index (3)

Beyond this lecture...

Test-Driven Development (TDD) with JUnit

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

Chen-Wei Wang

Learning Outcomes

This module is designed to help you learn about:

- **Testing** the Solution to a <u>Bounded Counter</u> Problem
- How <u>Manual</u>, Console Testers are Limited
- Deriving **Test Cases** for a <u>Bounded</u> Variable
- How <u>Automated</u>, JUnit Test Cases are Effective
- Test Driven Development (TDD) via **Regression Testing**

Consider two kinds of exceptions for a counter:

```
public class ValueTooLargeException extends Exception {
  ValueTooLargeException(String s) { super(s); }
}
public class ValueTooSmallException extends Exception {
  ValueTooSmallException(String s) { super(s); }
}
```

Any thrown object instantiated from these two classes must be handled (*catch-specify requirement*):

- Either *specify* throws ... in the method signature
  (i.e., propagating it to other caller)
- Or *handle* it in a try-catch block

Motivating Example: Two Types of Errors (2)

Approach 1 – Specify: Indicate in the method signature that a specific exception might be thrown.

Example 1: Method that throws the exception

```
class C1 {
  void m1(int x) throws ValueTooSmallException {
    if(x < 0) {
      throw new ValueTooSmallException("val " + x);
    }
  }
}
```

Example 2: Method that calls another which throws the exception

```
class C2 {
  C1 c1;
  void m2(int x) throws ValueTooSmallException {
    c1.m1(x);
  }
}
```

Approach 2 – Catch: Handle the thrown exception(s) in a try-catch block.

```
class C3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int x = input.nextInt();
    C2 c2 = new c2();
    try {
      c2.m2(x);
    }
    catch(ValueTooSmallException e) { ... }
  }
}
```

A Simple Counter (1)

Consider a class for keeping track of an integer counter value:

```java
public class Counter {
 public final static int MAX_VALUE = 3;
 public final static int MIN_VALUE = 0;
 private int value;
 public Counter() {
   this.value = Counter.MIN_VALUE;
 }
 public int getValue() {
   return value;
 }
 ... /* more later! */
```

- Access ***private*** attribute `value` using ***public*** accessor `getValue`.
- Two class-wide (i.e., `static`) constants (i.e., `final`) for lower and upper bounds of the counter value.
- Initialize the counter value to its lower bound.
- **Requirement** :

  The counter value must be between its lower and upper bounds.

Consider the two possible exceptional scenarios:

- An attempt to <u>increment</u> **above** the counter's upper bound.
- An attempt to <u>decrement</u> **below** the counter's lower bound.

A Simple Counter (2)

```
/* class Counter */
 public void increment() throws ValueTooLargeException {
   if(value == Counter.MAX_VALUE) {
    throw new ValueTooLargeException("value is " + value);
   }
   else { value ++; }
 }

 public void decrement() throws ValueTooSmallException {
   if(value == Counter.MIN_VALUE) {
    throw new ValueTooSmallException("value is " + value);
   }
   else { value --; }
 }
}
```

- Change the counter value via two mutator methods.
- Changes on the counter value may *trigger an exception*:
  - Attempt to **increment** when counter already reaches its **maximum**.
  - Attempt to **decrement** when counter already reaches its **minimum**.

Components of a Test

- Manipulate the relevant object(s).
  - e.g., *Initialize a counter object `c`, then call `c.increment()`.*
  - e.g., *Initialize a counter object `c`, then call `c.decrement()`.*
- What do you **expect to happen** ?
  - e.g., *value of counter is such that `Counter.MIN_VALUE + 1`*
  - e.g., *`ValueTooSmallException` is thrown*
- What does your program **actually produce** ?
  - e.g., *call `c.getValue()` to find out.*
  - e.g., *Use a `try-catch` block to find out* (to be discussed!).
- A test:
  - **Passes** if expected outcome occurs.
  - **Fails** if expected outcome does *not* occur.
- To start with, we develop tests via a **console tester** class (i.e., with the `main` method).

```
1  public class CounterTester1 {
2   public static void main(String[] args) {
3     Counter c = new Counter();
4     println("Init val: " + c.getValue());
5     try {
6       c.decrement();
7       println("Error: ValueTooSmallException NOT thrown.");
8     }
9     catch (ValueTooSmallException e) {
10      println("Success: ValueTooSmallException thrown.");
11    }
12   } /* end of main method */
13 } /* end of class CounterTester1 */
```

- **L3** sets c.value to 0.
- At **L6**, if method decrement is implemented:
  - **Correctly** ⇒ we expect a ValueTooSmallException.
    ⇒ Execution jumps to **L9**, **L10 – L12**, then the program terminates.
  - **Incorrectly** ⇒ expected ValueTooSmallException *wouldn't* occur.
    ⇒ Execution continues to **L7**, **L8**, **L12**, then the program terminates.

See the equivalent, automated JUnit test testDecFromMinValue.

```
1   public class CounterTester1 {
2     public static void main(String[] args) {
3       Counter c = new Counter();
4       println("Init val: " + c.getValue());
5       try {
6         c.decrement();
7         println("Error: ValueTooSmallException NOT thrown.");
8       }
9       catch (ValueTooSmallException e) {
10        println("Success: ValueTooSmallException thrown.");
11      }
12    } /* end of main method */
13  } /* end of class CounterTester1 */
```

- Say method `decrement` is implemented *correctly*.
- **Lines 3 – 6, 9 – 11, 12** executed, giving the Console Output:

```
Init val: 0
Success: ValueTooSmallException thrown.
```

- The real value of a **test** is:
  - Not only to confirm <u>when</u> your implementation is **correct**,
  - But also to reveal errors <u>when</u> your implementation is **incorrect**.

- Say now method `decrement` was implemented **incorrectly**:

```
class Counter {
  ...
  public void decrement() throws ValueTooSmallException {
    if(value < Counter.MIN_VALUE) {
      throw new ValueTooSmallException("value is " + value);
    }
    else { value --; }
  }
}
```

- Is the same console tester able to **reveal** this **incorrect** implementation?

Testing Counter via Console V1 (1.3.2)

```
1   public class CounterTester1 {
2     public static void main(String[] args) {
3       Counter c = new Counter();
4       println("Init val: " + c.getValue());
5       try {
6         c.decrement();
7         println("Error: ValueTooSmallException NOT thrown.");
8       }
9       catch (ValueTooSmallException e) {
10        println("Success: ValueTooSmallException thrown.");
11      }
12    } /* end of main method */
13  } /* end of class CounterTester1 */
```

- Say method `decrement` is implemented **_incorrectly_**.
- **Lines 3 – 6, 7 – 8, 12** executed, giving the Console Output:

```
Init val: 0
Error: ValueTooSmallException NOT thrown.
```

```
1   public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8        try {
9          c.increment();
10         println("Error: ValueTooLargeException NOT thrown.");
11       } /* end of inner try */
12       catch (ValueTooLargeException e) {
13         println("Success: ValueTooLargeException thrown.");
14       } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException thrown unexpectedly.");
18     } /* end of outer catch */
19   } /* end of main method */
20  } /* end of CounterTester2 class */
```

See the equivalent, automated JUnit test testIncFromMaxValue.

```
1  public class CounterTester2 {
2   public static void main(String[] args) {
3     Counter c = new Counter();
4     println("Current val: " + c.getValue());
5     try {
6       c.increment(); c.increment(); c.increment();
7       println("Current val: " + c.getValue());
8       try {
9         c.increment();
10        println("Error: ValueTooLargeException NOT thrown.");
11      } /* end of inner try */
12      catch (ValueTooLargeException e) {
13        println("Success: ValueTooLargeException thrown.");
14      } /* end of inner catch */
15     } /* end of outer try */
16     catch (ValueTooLargeException e) {
17       println("Error: ValueTooLargeException thrown unexpectedly.");
18     } /* end of outer catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

- Say method `increment` is implemented *correctly*.
- **Lines 3 – 9, 12 – 15, 19** executed, with Console Output:

```
Current val: 0
Current val: 3
Success: ValueTooLargeException thrown.
```

```
1   public class CounterTester2 {
2     public static void main(String[] args) {
3       Counter c = new Counter();
4       println("Current val: " + c.getValue());
5       try {
6         c.increment(); c.increment(); c.increment();
7         println("Current val: " + c.getValue());
8         try {
9           c.increment();
10          println("Error: ValueTooLargeException NOT thrown.");
11        } /* end of inner try */
12        catch (ValueTooLargeException e) {
13          println("Success: ValueTooLargeException thrown.");
14        } /* end of inner catch */
15      } /* end of outer try */
16      catch (ValueTooLargeException e) {
17        println("Error: ValueTooLargeException thrown unexpectedly.");
18      } /* end of outer catch */
19    } /* end of main method */
20  } /* end of CounterTester2 class */
```

○ **Exercise**: Give an *incorrect* method increment, so that
○ **Lines 3 – 6, 16 – 18, 19** executed, with Console Output:

```
Current val: 0
Error: ValueTooLargeException was thrown unexpectedly.
```

```
1  public class CounterTester2 {
2   public static void main(String[] args) {
3     Counter c = new Counter();
4     println("Current val: " + c.getValue());
5     try {
6       c.increment(); c.increment(); c.increment();
7       println("Current val: " + c.getValue());
8       try {
9         c.increment();
10        println("Error: ValueTooLargeException NOT thrown.");
11      } /* end of inner try */
12      catch (ValueTooLargeException e) {
13        println("Success: ValueTooLargeException thrown.");
14      } /* end of inner catch */
15    } /* end of outer try */
16    catch (ValueTooLargeException e) {
17      println("Error: ValueTooLargeException thrown unexpectedly.");
18    } /* end of outer catch */
19   } /* end of main method */
20  } /* end of CounterTester2 class */
```

- **Exercise**: Give an *incorrect* method increment, so that
- **Lines 3 – 11, 15, 19** executed, with Console Output:

```
Current val: 0
Current val: 3
Error: ValueTooLargeException was NOT thrown.
```

Q. Can this alternative to `ConsoleTester2` work (<u>without</u> nested `try-catch`)?

```
1  public class CounterTester2 {
2    public static void main(String[] args) {
3      Counter c = new Counter();
4      println("Current val: " + c.getValue());
5      try {
6        c.increment(); c.increment(); c.increment();
7        println("Current val: " + c.getValue());
8      }
9      catch (ValueTooLargeException e) {
10       println("Error: ValueTooLargeException thrown unexpectedly.");
11     }
12     try {
13       c.increment();
14       println("Error: ValueTooLargeException NOT thrown.");
15     } /* end of inner try */
16     catch (ValueTooLargeException e) {
17       println("Success: ValueTooLargeException thrown.");
18     } /* end of inner catch */
19   } /* end of main method */
20 } /* end of CounterTester2 class */
```

A. Say one of the first 3 `c.increment()` *mistakenly* throws VTLE.

- After **L10** is executed, flow of execution *still continues* to **L12**.
- This allows the 4th `c.increment` to be executed!
- Constrast this with the structurally-similar JUnit test `testIncFromMaxValue`: here.

Testing Counter via Console (V3)

```java
import java.util.Scanner;
public class CounterTester3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    String cmd = null; Counter c = new Counter();
    boolean userWantsToContinue = true;
    while(userWantsToContinue) {
      println("Enter \"inc\", \"dec\", or \"val\":");
      cmd = input.nextLine();
      try {
       if(cmd.equals("inc")) { c.increment(); }
       else if(cmd.equals("dec")) { c.decrement(); }
       else if(cmd.equals("val")) { println( c.getValue() ); }
       else { userWantsToContinue = false; println("Bye!"); }
      } /* end of try */
      catch(ValueTooLargeException e){ println("Value too big!"); }
      catch(ValueTooSmallException e){ println("Value too small!"); }
    } /* end of while */
  } /* end of main method */
} /* end of class CounterTester3 */
```

Test Case 1: Decrement when the counter value is too small.

```
Enter "inc", "dec", or "val":
val
0
Enter "inc", "dec", or "val":
dec
Value too small!
Enter "inc", "dec", or "val":
exit
Bye!
```

Test Case 2: Increment when the counter value is too big.

```
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
val
3
Enter "inc", "dec", or "val":
inc
Value too big!
Enter "inc", "dec", or "val":
exit
Bye!
```

Limitations of Testing from the Console

- Do **Test Cases** 1 & 2 suffice to test `Counter`'s *correctness*?
  - Is it plausible to claim that the implementation of `Counter` is *correct* because it passes the two test cases?
- What other test cases can you think of?

c.getValue()	c.increment()	c.decrement()
0	1	ValueTooSmall
1	2	0
2	3	1
3	ValueTooLarge	2

- So in total we need 8 test cases. ⇒ 6 more separate
  - `CounterTester` classes to create (like `CounterTester1`)!
  - Console interactions with `CounterTester3`!
- Problems? It is <u>inconvenient</u> and ***error-prone*** to:
  - Run each TC by executing `main` of a `CounterTester` and comparing console outputs ***with your eyes***.
  - ***Re-run*** **manually** all TCs whenever `Counter` is changed.

    Regression Testing : Any **change** introduced to your software *must not compromise* its established ***correctness***.

Why JUnit?

- *Automate* the ***testing of correctness*** of your Java classes.
- Once you derive the list of tests, translate it into a JUnit test case, which is just a Java class that you can execute upon.
- JUnit tests are *helpful callers/clients* of your classes, where each test may:
  - Either attempt to use a method in a ***legal*** way (i.e., ***satisfying*** its precondition), and report:
    - ***Success*** if the result is as expected
    - ***Failure*** if the result is ***not*** as expected
  - Or attempt to use a method in an ***illegal*** way (i.e., ***not satisfying*** its precondition), and report:
    - ***Success*** if the expected exception (e.g., `ValueTooSmallException`) occurs.
    - ***Failure*** if the expected exception does ***not*** occur.

Step 1:

○ In Eclipse, create a Java project `ExampleTestingCounter`

○ *Separation of concerns* :

- Group classes for ***implementation*** (i.e., `Counter`)
  into package `implementation`.
- Group classes classes for ***testing*** (to be created)
  into package `tests`.
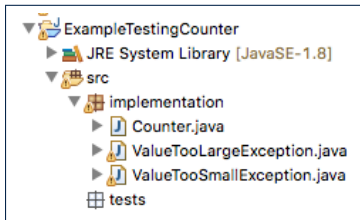
How to Use JUnit: New JUnit Test Case (1)

Step 2: Create a new <mark>*JUnit Test Case*</mark> in `tests` package.
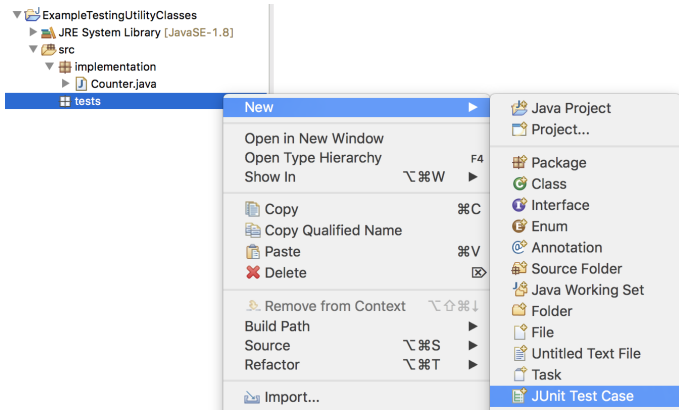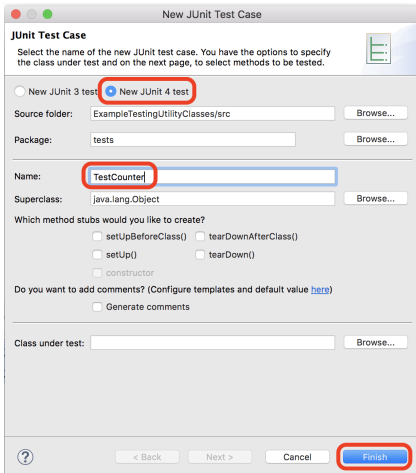
Create one JUnit Test Case to test one Java class only.

⇒ If you have *n Java classes to test*, create *n JUnit test cases*.

How to Use JUnit: New JUnit Test Case (2)

Step 3: <u>Select</u> the version of JUnit (JUnit 4); <u>Enter</u> the name of test case (`TestCounter`); <u>Finish</u> creating the new test case.

How to Use JUnit: Adding JUnit Library

Upon creating the very first test case, you will be prompted to add the JUnit library to your project's build path.
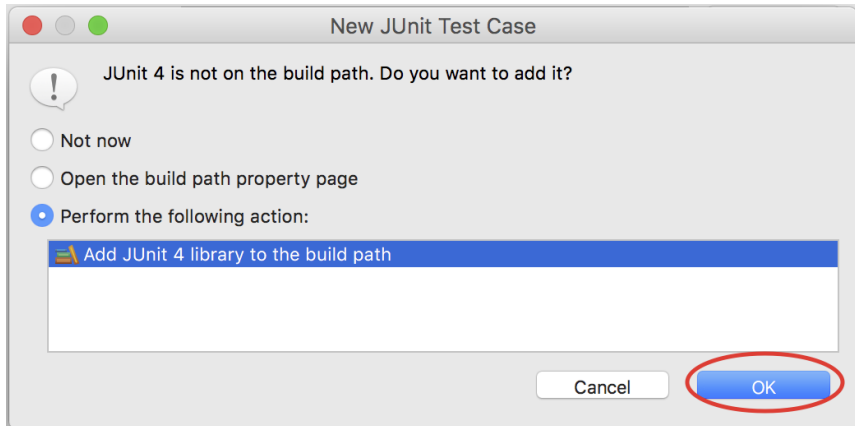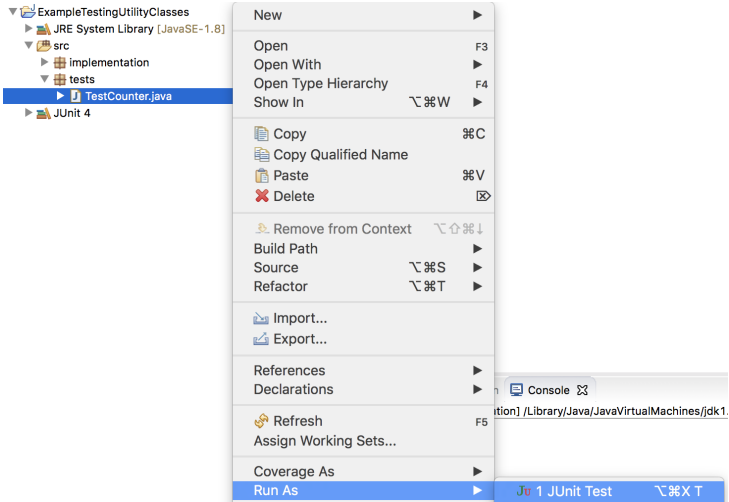
```
TestCounter.java ⊠
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7         fail("Not yet implemented");
8     }
9 }
```

○ **Lines 6 – 8**: `test` is just an ***ordinary mutator method*** that has a one-line implementation body.

○ **Line 5** is critical: Prepend the tag ***@Test*** verbatim, requiring that ***the method is to be treated as a JUnit test***.
⇒ When `TestCounter` is run as a JUnit Test Case, only ***those methods prepended by the @Test tags*** will be run and reported.

○ **Line 7**: By default, we deliberately fail the test with a message "Not yet implemented".

How to Use JUnit: Running Test Case

Step 4: Run the `TestCounter` class as a JUnit Test.

How to Use JUnit: Generating Test Report

A *report* is generated after running all tests (i.e., methods prepended with **@Test**) in `TestCounter`.

How to Use JUnit: Interpreting Test Report

- A **_test_** is a method prepended with the **`@Test`** tag.
- The result of running a test is considered:
  - **_Failure_** if either
    - an assertion failure (e.g., caused by `fail`, `assertTrue`, `assertEquals`) occurs; or
    - an _unexpected_ exception (e.g., `NullPointerException`, `ArrayIndexOutOfBoundException`) is thrown.
  - **_Success_** if neither assertion failures nor _unexpected_ exceptions occur.
- After running all tests:
  - A _green_ bar means that **all** tests succeed.
    ⇒ Keep challenging yourself if more tests may be added.
  - A _red_ bar means that **at least one** test fails.
    ⇒ Keep fixing the class under test and re-runing all tests, until you receive a _green_ bar.
- **Question**: What is the easiest way to making `test` a _success_?
  Answer: Delete the call `fail("Not yet implemented")`.

```
TestCounter.java ⊠
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7 //       fail("Not yet implemented");
8     }
9 }
```

Now, the body of `test` simply does nothing.
⇒ Neither assertion failures nor exceptions will occur.
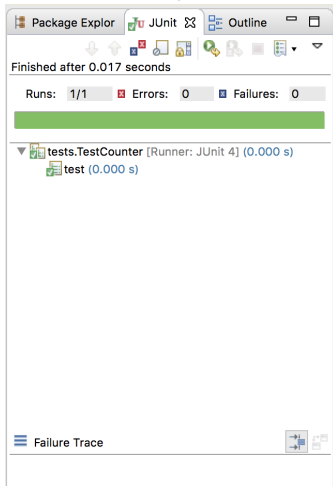⇒ The execution of `test` will be considered as a *success*.

∵ There is currently only one test in `TestCounter`.
∴ We will receive a *green* bar!
Caution: `test` which passes at the moment is **not useful** at all!

How to Use JUnit: Re-Running Test Case

A new report is generated after re-running all tests (i.e., methods prepended with `@Test`) in `TestCounter`.

- Recall the complete list of cases for testing `Counter`:

c.getValue()	c.increment()	c.decrement()
0	1	ValueTooSmall
1	2	0
2	3	1
3	ValueTooLarge	2

- Let's turn the two cases in the 1st row into two JUnit tests:
  - Test for the *green* cell *succeeds* if:
    - No failures and exceptions occur; and
    - The new counter value is `1`.
  - Tests for *red* cells *succeed* if the *expected exceptions* occur
    (`ValueTooSmallException` & `ValueTooLargeException`).
  - Common JUnit assertion methods:
    - **void assertNull**(**Object** *o*)
    - **void assertEquals**(**int** *expected*, **int** *actual*)
    - **void assertEquals**(**double** *exp*, **double** *act*, **double** *epsilon*)
    - **void assertArrayEquals**(*expected*, *actuals*)
    - **void assertTrue**(**boolean** *condition*)
    - **void fail**(**String** *message*)

JUnit Assertions: Examples (1)

Consider the following class:

```java
public class Point {
  private int x; private int y;
  public Point(int x, int y) { this.x = x; this.y = y; }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
}
```

Then consider these assertions. Do they *pass* or *fail*?

```java
Point p;
assertNull(p);   ✓
assertTrue(p == null);   ✓
assertFalse(p != null);   ✓
assertEquals(3, p.getX());   × /* NullPointerException */
p = new Point(3, 4);
assertNull(p);   ×
assertTrue(p == null);   ×
assertFalse(p != null);   ×
assertEquals(3, p.getX());   ✓
assertTrue(p.getX() == 3 && p.getY() == 4);   ✓
```

JUnit Assertions: Examples (2)

- Consider the following class:

```java
public class Circle {
  private double radius;
  public Circle(double radius) { this.radius = radius; }
  public int getArea() { return 3.14 * radius * radius; }
}
```

- How do we test `c.getArea()`?
  - Mathematically: $3.4 \times 3.4 \times 3.14 = 36.2984$
  - However, base-10 numbers *cannot* be represented perfectly in the binary format.
  - When comparing fractional numbers, allow some *tolerance* :

$$36.2984 - \textbf{0.01} \leq c.getArea() \leq 36.2984 + \textbf{0.01}$$

- Then consider these assertions. Do they *pass* or *fail*?

```java
Circle c = new Circle(3.4);
assertTrue(36.2984, c.getArea(), 0.01);  ✓
```

How to Use JUnit: Assertion Methods

method name / parameters	description
`assertTrue(test)` `assertTrue("message", test)`	Causes this test method to fail if the given `boolean` test is not `true`.
`assertFalse(test)` `assertFalse("message", test)`	Causes this test method to fail if the given `boolean` test is not `false`.
`assertEquals(expectedValue, value)` `assertEquals("message", expectedValue, value)`	Causes this test method to fail if the given two values are not equal to each other. (For objects, it uses the `equals` method to compare them.) The first of the two values is considered to be the result that you expect; the second is the actual result produced by the class under test.
`assertNotEquals(value1, value2)` `assertNotEquals("message", value1, value2)`	Causes this test method to fail if the given two values *are* equal to each other. (For objects, it uses the `equals` method to compare them.)
`assertNull(value)` `assertNull("message", value)`	Causes this test method to fail if the given value is not `null`.
`assertNotNull(value)` `assertNotNull("message", value)`	Causes this test method to fail if the given value *is* `null`.

`assertSame(expectedValue, value)`
`assertSame("message", expectedValue, value)`
`assertNotSame(value1, value2)`
`assertNotSame("message", value1, value2)`
	Identical to `assertEquals` and `assertNotEquals` respectively, except that for objects, it uses the `==` operator rather than the `equals` method to compare them. (The difference is that two objects that have the same state might be `equals` to each other, but not `==` to each other. An object is only `==` to itself.)

`fail()` `fail("message")`	Causes this test method to fail.

```
1   @Test
2   public void testIncAfterCreation() {
3     Counter c = new Counter();
4     assertEquals(Counter.MIN_VALUE, c.getValue());
5     try {
6       c.increment();
7       assertEquals(1, c.getValue());
8     }
9     catch(ValueTooBigException e) {
10      /* Exception is not expected to be thrown. */
11      fail("ValueTooBigException is not expected.");
12    }
13  }
```

- **Line 6** requires a try-catch block ∵ potential *ValueTooBigException*
- **Lines 4, 7 11** are all assertions:
  - **Lines 4 & 7** assert that *c.getValue()* returns the expected values.
  - **Line 11**: an assertion failure ∵ unexpected *ValueTooBigException*
- **Line 7** can be rewritten as **assertTrue**(1 == c.getValue()).

How to Use JUnit: Adding More Tests (2.2)

- Don't lose the big picture!
- JUnit test in previous slide automates this console interaction:

```
Enter "inc", "dec", or "val":
val
0
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
val
1
Enter "inc", "dec", or "val":
exit
Bye!
```

- ***Automation*** *is exactly rationale behind using JUnit!*

```
1   @Test
2   public void testDecFromMinValue() {
3     Counter c = new Counter();
4     assertEquals(Counter.MIN_VALUE, c.getValue());
5     try {
6       c.decrement();
7       fail ("ValueTooSmallException is expected.");
8     }
9     catch(ValueTooSmallException e) {
10      /* Exception is expected to be thrown. */
11    }
12  }
```

- **Line 6** requires a try-catch block ∵ potential *ValueTooBigException*
- **Lines 4 & 7** are both assertions:
  - **Lines 4** asserts that *c.getValue()* returns the expected value (i.e., *Counter.MIN_VALUE*).
  - **Line 7**: an assertion failure ∵ expected *ValueTooSmallException* not thrown

How to Use JUnit: Adding More Tests (3.2)

- Again, don't lose the big picture!

  Automation is exactly rationale behind using JUnit!

- JUnit test in previous slide automates `CounterTester1` and
  the following console interaction for `CounterTester3`:

```
Enter "inc", "dec", or "val":
val
0
Enter "inc", "dec", or "val":
dec
Value too small!
Enter "inc", "dec", or "val":
exit
Bye!
```

```
1   @Test
2   public void testIncFromMaxValue() {
3     Counter c = new Counter();
4     try {
5       c.increment(); c.increment(); c.increment();
6     }
7     catch (ValueTooLargeException e) {
8       fail("ValueTooLargeException was thrown unexpectedly.");
9     }
10    assertEquals(Counter.MAX_VALUE, c.getValue());
11    try {
12      c.increment();
13      fail("ValueTooLargeException was NOT thrown as expected.");
14    }
15    catch (ValueTooLargeException e) {
16      /* Do nothing: ValueTooLargeException thrown as expected. */
17    }
18  }
```

○ **L4 – L9**: a VTLE *is not* expected; **L11 – 17**: a VTLE *is* expected.

See the equivalent, manual `ConsoleTester2`. Contrast with the alternative to `ConsoleTester2`.

```
1   @Test
2   public void testIncFromMaxValue() {
3     Counter c = new Counter();
4     try {
5       c.increment(); c.increment(); c.increment();
6     }
7     catch (ValueTooLargeException e) {
8       fail("ValueTooLargeException was thrown unexpectedly.");
9     }
10    assertEquals(Counter.MAX_VALUE, c.getValue());
11    try {
12      c.increment();
13      fail("ValueTooLargeException was NOT thrown as expected.");
14    }
15    catch (ValueTooLargeException e) {
16      /* Do nothing: ValueTooLargeException thrown as expected. */
17    }
18  }
```

Contrast with the structurally-similar console tester.

- JUnit test in previous slide *automates* `CounterTester2` and the following console interaction for `CounterTester3`:

```
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
inc
Enter "inc", "dec", or "val":
val
3
Enter "inc", "dec", or "val":
inc
Value too big!
Enter "inc", "dec", or "val":
exit
Bye!
```

Q: Can we rewrite `testIncFromMaxValue` to:

```
1   @Test
2   public void testIncFromMaxValue() {
3     Counter c = new Counter();
4     try {
5       c.increment();
6       c.increment();
7       c.increment();
8       assertEquals(Counter.MAX_VALUE, c.getValue());
9       c.increment();
10      fail("ValueTooLargeException was NOT thrown as expected.");
11    }
12    catch (ValueTooLargeException e) { }
13  }
```

No!
At **Line 12**, we would not know which line throws the VTLE:
- If it was any of the calls in **L5 – L7**, then it's *not right*.
- If it was **L9**, then it's *right*.

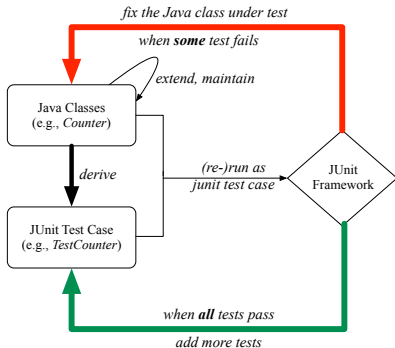Loops can make it effective on generating test cases:

```
1  @Test
2  public void testIncDecFromMiddleValues() {
3    Counter c = new Counter();
4    try {
5      for(int i = Counter.MIN_VALUE; i < Counter.MAX_VALUE; i ++) {
6        int currentValue = c.getValue();
7        c.increment();
8        assertEquals(currentValue + 1, c.getValue());
9      }
10     for(int i = Counter.MAX_VALUE; i > Counter.MIN_VALUE; i --) {
11       int currentValue = c.getValue();
12       c.decrement();
13       assertEquals(currentValue - 1, c.getValue());
14     }
15   }
16   catch(ValueTooLargeException e) {
17     fail("ValueTooLargeException is thrown unexpectedly");
18   }
19   catch(ValueTooSmallException e) {
20     fail("ValueTooSmallException is thrown unexpectedly");
21   }
22 }
```

Exercises

1. Run all 8 tests and make sure you receive a *green* bar.
2. Now, introduction an error to the implementation: Change the line `value ++` in `Counter.increment` to `--`.
   - Re-run all 8 tests and you should receive a *red* bar. [Why?]
   - Undo the ***injection of error***, and re-run all 8 tests. [What happens?]

Test-Driven Development (TDD)

fix the Java class under test

*when **some** test fails*

extend, maintain

Java Classes
(e.g., *Counter*)

derive

*(re-)run as
junit test case*

JUnit
Framework

JUnit Test Case
(e.g., *TestCounter*)

*when **all** tests pass*

add more tests

Maintain a collection of tests which define the ***correctness*** of your
Java class under development (CUD):

- Derive and run tests as soon as your CUD is *testable* .
  i.e., A Java class is testable when defined with method signatures.
- ***Red*** bar reported: Fix the class under test (CUT) until *green* bar.
- ***Green*** bar reported: Add more tests and Fix CUT when necessary.

Resources

- Official Site of JUnit 4:

  http://junit.org/junit4/

- API of JUnit assertions:

  http://junit.sourceforge.net/javadoc/org/junit/Assert.html

- Another JUnit Tutorial example:

  https://courses.cs.washington.edu/courses/cse143/11wi/

  eclipse-tutorial/junit.shtml

Play with the source code `ExampleTestingCounter.zip`
Tip. Change input values so as to explore, in Eclipse ***debugger***,
possible (***normal*** vs. ***abnormal***) ***execution paths*** .

Index (1)

Index (2)

Index (3)

Index (4)

Object Equality and Call by Value

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

- ***Object equality***: To ***Override*** or ***Not*** to Override
- Asserting ***Object Equality***: `assertSame` vs. `assertEquals`
- Short-Circuit Effect (SCE): `&&` vs. `||`
- Equality for Array-, Reference-Typed Attributes
- ***Call by Value***: Primitive vs. Reference Argument Values

Equality (1)

- Recall that
  - A *primitive* variable stores a primitive *value*.
    e.g., `double d1 = 7.5; double d2 = 7.5;`
  - A *reference* variable stores the *address* to some object (rather than storing the object itself).
    e.g., `Point p1 = new Point(2, 3)` assigns to `p1` the address of the new `Point` object
    e.g., `Point p2 = new Point(2, 3)` assigns to `p2` the address of <u>another</u> new `Point` object
- The binary operator `==` may be applied to compare:
  - *Primitive* variables: their *values* are compared
    e.g., `d1 == d2` evaluates to **true**
  - *Reference* variables: the *addresses* they store are compared (**rather than** comparing contents of the objects they refer to)
    e.g., `p1 == p2` evaluates to **false** because `p1` and `p2` are addresses of <u>different</u> objects, even if their contents are <u>identical</u>.

Equality (2.1)

- Implicitly:
  - Every class is a ***child/sub*** class of the **Object** class.
  - The **Object** class is the ***parent/super*** class of every class.
- There is a useful <u>accessor method</u> that every class ***inherits*** from the **Object** class:
  - | **public boolean equals**(**Object** *obj*) |
    - Indicates whether some other object <u>obj</u> is "equal to" this one.
    - The <u>default</u> definition inherited from Object:

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

      e.g., Say p1 and p2 are of type <u>Point**V1**</u>
      in which the equals method is <u>not</u> ***redefined***/***overridden***,
      then p1.**equals**(p2) boils down to (p1 **==** p2).
  - Very often when you define new classes, you want to
    redefine / *override* the inherited definition of equals.

LASSONDE
SCHOOL OF ENGINEERING

```
int i = 10;
int j = 12;
boolean sameValue = i.equals(j);
```

Compilation Error

The equals method is only applicable to reference types.

Fix

Write i == j instead.

```
public class PointV1 {
  private int x; private int y;
  public PointV1(int x, int y) { this.x = x; this.y = y; }
}
```

```
1  String s = "(2, 3)";
2  PointV1 p1 = new PointV1(2, 3);
3  PointV1 p2 = new PointV1(2, 3);
4  PointV1 p3 = new PointV1(4, 6);
5  System.out.println(p1 == p2);  /* false */
6  System.out.println(p2 == p3);  /* false */
7  System.out.println(p1.equals(p1));  /* true */
8  System.out.println(p1.equals(null));  /* false */
9  System.out.println(p1.equals(s));  /* false */
10 System.out.println(p1.equals(p2));  /* false */
11 System.out.println(p2.equals(p3));  /* false */
```

- The equals method is not explicitly redefined/overridden in class
  <u>PointV1</u> ⇒ The <u>default version</u> inherited from class **Object** is called.
  e.g., Executing p1.**equals**(null) boils down to (p1 **==** null).
- To compare contents of <u>PointV1</u> objects, *redefine*/*override* equals.

Requirements of `equals`

Given that **_reference variables_** x, y, z are not `null`:

- $$\neg\, x.equals(null)$$

- _Reflexive_ :
  $$x.equals(x)$$

- _Symmetric_
  $$x.equals(y) \iff y.equals(x)$$

- _Transitive_
  $$x.equals(y) \wedge y.equals(z) \Rightarrow x.equals(z)$$

Equality (4.1)

To compare *contents* rather than addresses, override `equals`.

```java
public class PointV2 {
  private int x; private int y;
  public boolean equals (Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    PointV2 other = (PointV2) obj;
    return this.x == other.x && this.y == other.y;
  }
}
```

```java
1   String s = "(2, 3)";
2   PointV2 p1 = new PointV2(2, 3);
3   PointV2 p2 = new PointV2(2, 3);
4   PointV2 p3 = new PointV2(4, 6);
5   System.out.println(p1 == p2); /* false */
6   System.out.println(p2 == p3); /* false */
7   System.out.println(p1.equals(p1)); /* true */
8   System.out.println(p1.equals(null)); /* false */
9   System.out.println(p1.equals(s)); /* false */
10  System.out.println(p1.equals(p2)); /* true */
11  System.out.println(p2.equals(p3)); /* false */
```

Equality (4.2)

- When making a method call `p.equals(o)`:
  - Say variable `p` is declared of type <u>Point**V2**</u>
  - Variable `o` can be declared of any type (e.g., <u>Point**V2**</u>, String)
- We define `p` and `o` as *equal* if:
  - Either `p` and `o` refer to the same object;
  - Or:
    - `o` does **not** store the **null** address.
    - `p` and `o` at runtime point to objects of the same type.
    - The `x` and `y` coordinates are the same.
- **Q**: In the `equals` method of Point, why is there no such a line:

```
class PointV2 {
  public boolean equals(Object obj) {
    if(this == null) { return false; }
```

A: If `this` was null, a ***NullPointerException*** would have occurred, preventing the body of `equals` from being executed.

Equality (4.3)

```
1  public class PointV2 {
2    public boolean equals(Object obj) {
3      ...
4      if(this.getClass() != obj.getClass()) { return false; }
5      PointV2 other = (PointV2) obj;
6      return this.x == other.x && this.y == other.y;
7    }
8  }
```

- $\boxed{\texttt{Object obj}}$ at **L2** declares a parameter obj of type Object.

- $\boxed{\texttt{PointV2 other}}$ at **L5** declares a variable p of type PointV2.
  We call such types declared at **compile time** as *static type*.

- *Applicable attributes/methods* callable upon a variable depends on its *static type*.
  e.g., We may only call the small list of methods defined in Object class on obj, which does not include x and y (specific to PointV2).

- If we are **certain** that an object's "actual" type is different from its *static type*, then we can $\boxed{cast}$ it.
  e.g., Given that this.getClass() == obj.getClass(), we are sure that obj is also a Point, so we can cast it to PointV2.

- The *cast* $\boxed{\texttt{(PointV2) obj}}$ creates an **alias** of obj, upon which (or upon its alias such as other) more methods can be invoked.

Equality (5)

Two notions of **equality** for variables of **reference** types:

- **Reference Equality** : use == to compare **addresses**

- **Object Equality** : define `equals` method to compare **contents**

```
1  PointV2 p1 = new PointV2(3, 4);
2  PointV2 p2 = new PointV2(3, 4);
3  PointV2 p3 = new PointV2(4, 5);
4  System.out.println(p1 == p1);  /* true */
5  System.out.println(p1.equals(p1));  /* true */
6  System.out.println(p1 == p2);  /* false */
7  System.out.println(p1.equals(p2));  /* true */
8  System.out.println(p2 == p3);  /* false */
9  System.out.println(p2.equals(p3));  /* false */
```

- Being *reference*-equal implies being *object*-equal.
- Being *object*-equal does **not** imply being *reference*-equal.

Equality in JUnit (1.1)

- **assertSame**(exp1, exp2)
  - Passes if exp1 and exp2 are references to the same object
    - ≈ **assertTrue**(exp1 == exp2)
    - ≈ **assertFalse**(exp1 != exp2)

```
PointV1 p1 = new PointV1(3, 4);
PointV1 p2 = new PointV1(3, 4);
PointV1 p3 = p1;
assertSame(p1, p3);  ✓
assertSame(p2, p3);  ×
```

- **assertEquals**(exp1, exp2)
  - ≈ $\boxed{\text{exp1 == exp2}}$ if exp1 and exp2 are **primitive** type

```
int i = 10;
int j = 20;
assertEquals(i, j);  ×
```

Equality in JUnit (1.2)

- **_assertEquals_**(exp1, exp2)

  ○ ≈ | exp1.**equals**(exp2) | if exp1 and exp2 are **_reference_** type

     Case 1: If equals is **_not_** explicitly overridden in exp1's dynamic type
     ≈ **_assertSame_**(exp1, exp2)

     ```
     PointV1 p1 = new PointV1(3, 4);
     PointV1 p2 = new PointV1(3, 4);
     PointV2 p3 = new PointV2(3, 4);
     assertEquals(p1, p2);  × /* ∵ different PointV1 objects */
     assertEquals(p2, p3);  × /* ∵ different object addresses */
     ```

     Case 2: If equals is explicitly **_overridden_** in exp1's dynamic type
     ≈ exp1.**_equals_**(exp2)

     ```
     PointV1 p1 = new PointV1(3, 4);
     PointV1 p2 = new PointV1(3, 4);
     PointV2 p3 = new PointV2(3, 4);
     assertEquals(p1, p2);  × /* ≈ p1.equals(p2) ≈ p1 == p2 */
     assertEquals(p2, p3);  × /* ≈ p2.equals(p3) ≈ p2 == p3 */
     assertEquals(p3, p2);  × /* ≈ p3.equals(p2) ≈ p3.getClass()==p2.getClass() */
     ```

Equality in JUnit (2)

```
@Test
public void testEqualityOfPointV1() {
  PointV1 p1 = new PointV1(3, 4); PointV1 p2 = new PointV1(3, 4);
  assertFalse(p1 == p2); assertFalse(p2 == p1);
  /* assertSame(p1, p2); assertSame(p2, p1); */ /* both fail */
  assertFalse(p1.equals(p2)); assertFalse(p2.equals(p1));
  assertTrue(p1.getX() == p2.getX() && p1.getY() == p2.getY());
}
@Test
public void testEqualityOfPointV2() {
  PointV2 p3 = new PointV2(3, 4); PointV2 p4 = new PointV2(3, 4);
  assertFalse(p3 == p4); assertFalse(p4 == p3);
  /* assertSame(p3, p4); assertSame(p4, p3); */ /* both fail */
  assertTrue(p3.equals(p4)); assertTrue(p4.equals(p3));
  assertEquals(p3, p4); assertEquals(p4, p3);
}
@Test
public void testEqualityOfPointV1andPointv2() {
  PointV1 p1 = new PointV1(3, 4); PointV2 p2 = new PointV2(3, 4);
  /* These two assertions do not compile because p1 and p2 are of different types. */
  /* assertFalse(p1 == p2); assertFalse(p2 == p1); */
  /* assertSame can take objects of different types and fail. */
  /* assertSame(p1, p2); */ /* compiles, but fails */
  /* assertSame(p2, p1); */ /* compiles, but fails */
  /* version of equals from Object is called */
  assertFalse(p1.equals(p2));
  /* version of equals from PointP2 is called */
  assertFalse(p2.equals(p1));
}
```

Equality (6.1)

Exercise: Persons are *equal* if names and measures are equal.

```
1  public class Person {
2    private String firstName; private String lastName;
3    private double weight; private double height;
4    public boolean equals(Object obj) {
5      if(this == obj) { return true; }
6      if(obj == null || this.getClass() != obj.getClass()) { return false; }
7      Person other = (Person) obj;
8      return
9          this.weight == other.weight
10        && this.height == other.height
11        && this.firstName.equals(other.firstName)
12        && this.lastName.equals(other.lastName);
13    }
14  }
```

Q: At **L6**, will we get a ***NullPointerException*** if obj is **null**?

A: ***No*** ∵ Short-Circuit Effect of ||

   obj is null, then obj == null evaluates to ***true***

   ⇒ no need to evaluate the RHS

The left operand obj == null acts as a <mark>*guard constraint*</mark> for the

right operand this.getClass() != obj.getClass().

Exercise: Persons are *equal* if names and measures are equal.

```java
public class Person {
  private String firstName; private String lastName;
  private double weight; private double height;
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null || this.getClass() != obj.getClass()) { return false; }
    Person other = (Person) obj;
    return
        this.weight == other.weight
     && this.height == other.height
     && this.firstName.equals(other.firstName)
     && this.lastName.equals(other.lastName);
  }
}
```

Q: At **L6**, if swapping the order of two operands of disjunction:
`this.getClass() != obj.getClass() || obj == null`
Will we get a *NullPointerException* if `obj` is **null**?

A: *Yes* ∵ Evaluation of operands is from left to right.

Equality (6.3)

Exercise: Persons are *equal* if names and measures are equal.

```
1   public class Person {
2     private String firstName; private String lastName;
3     private double weight; private double height;
4     public boolean equals(Object obj) {
5       if(this == obj) { return true; }
6       if(obj == null || this.getClass() != obj.getClass()) { return false; }
7       Person other = (Person) obj;
8       return
9            this.weight == other.weight
10        && this.height == other.height
11        && this.firstName.equals(other.firstName)
12        && this.lastName.equals(other.lastName);
13     }
14   }
```

Q: At **L11 & L12**, where is the `equals` method defined?

A: The `equals` method *overridden* in the `String` class.

When implementing the `equals` method for your own class, *reuse*
the `equals` methods *overridden* in other classes wherever possible.

Equality (6.4)

Person collectors are equal if containing equal lists of persons.

```
class PersonCollector {
 private Person[] persons;
 private int nop; /* number of persons */
 public PersonCollector() { ... }
 public void addPerson(Person p) { ... }
 public int getNop() { return this.nop; }
 public Person[] getPersons() { ... }
}
```

Redefine/*Override* the equals method in PersonCollector.

```
1  public boolean equals(Object obj) {
2    if(this == obj) { return true; }
3    if(obj == null || this.getClass() != obj.getClass()) { return false; }
4    PersonCollector other = (PersonCollector) obj;
5    boolean equal = false;
6    if(this.nop == other.nop) {
7      equal = true;
8      for(int i = 0; equal && i < this.nop; i ++) {
9        equal = this.persons[i].equals(other.persons[i]);
10     }
11   }
12   return equal;
13 }
```

Equality in JUnit (3)

```java
@Test
public void testPersonCollector() {
  Person p1 = new Person("A", "a", 180, 1.8);
  Person p2 = new Person("A", "a", 180, 1.8);
  Person p3 = new Person("B", "b", 200, 2.1);
  Person p4 = p3;
  assertFalse(p1 == p2); assertTrue(p1.equals(p2));
  assertTrue(p3 == p4); assertTrue(p3.equals(p4));
  PersonCollector pc1 = new PersonCollector();
  PersonCollector pc2 = new PersonCollector();
  assertFalse(pc1 == pc2); assertTrue(pc1.equals(pc2));
  pc1.addPerson(p1);
  assertFalse(pc1.equals(pc2));
  pc2.addPerson(p2);
  assertFalse(pc1.getPersons()[0] == pc2.getPersons()[0]);
  assertTrue(pc1.getPersons()[0].equals(pc2.getPersons()[0]));
  assertTrue(pc1.equals(pc2));
  pc1.addPerson(p3);
  pc2.addPerson(p4);
  assertTrue(pc1.getPersons()[1] == pc2.getPersons()[1]);
  assertTrue(pc1.getPersons()[1].equals(pc2.getPersons()[1]));
  assertTrue(pc1.equals(pc2));
  pc1.addPerson(new Person("A", "a", 175, 1.75));
  pc2.addPerson(new Person("A", "a", 165, 1.55));
  assertFalse(pc1.getPersons()[2] == pc2.getPersons()[2]);
  assertFalse(pc1.getPersons()[2].equals(pc2.getPersons()[2]));
  assertFalse(pc1.equals(pc2));
}
```

Beyond this lecture...

LASSONDE

- Play with the source code
  `ExampleEqualityPointsPersons.zip`
  Tip. Use the debugger to step into executing the
  various versions of `equals` method.
- Go back to your Review Tutorial: Extend the `Product`, `Entry`,
  and `RefurbishedStore` classes by *__overridden__* versions of
  the `equals` method.

Index (1)

Index (2)

Aggregation and Composition

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

Chen-Wei Wang

Learning Outcomes

This module is designed to help you learn about:

- ***Call by Value***: Primitive vs. Reference Argument Values
- Aggregation vs. Composition: ***Terminology*** and ***Modelling***
- ***Aggregation***: Building Sharing Links & ***Navigating*** Objects
- ***Composition***: Implementation via ***Copy Constructors***
- ***Design Decision*** : Aggregation or Composition?

Call by Value (1)

- Consider the general form of a call to some ***mutator method*** m, with ***context object*** co and **argument value** arg:

$$co.m(arg)$$

  - Argument variable **arg** is ***not*** passed directly to the method call.
  - Instead, argument variable **arg** is passed **indirectly**: a *copy* of the value stored in **arg** is made and passed to the method call.

- What can be the type of variable **arg**? [Primitive or Reference]
  - **arg** is primitive type (e.g., int, char, boolean, *etc.*):
    Call by Value : Copy of **arg**'s *stored value*
    (e.g., 2, 'j', true) is made and passed.
  - **arg** is reference type (e.g., String, Point, Person, *etc.*):
    Call by Value : Copy of **arg**'s *stored reference/address*
    (e.g., Point@5cb0d902) is made and passed.

Call by Value (2.1)

For illustration, let's assume the following variant of the `Point` class:

```java
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y){ this.y += y; }
  public void moveHorizontally(int x){ this.x += x; }
}
```

Call by Value (2.2.1)

```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByVal() {
3    Util u = new Util();
4    int i = 10;
5    assertTrue(i == 10);
6    u.reassignInt(i);
7    assertTrue(i == 10);
8  }
```

- **Before** the mutator call at **L6**, *primitive* variable `i` stores 10.

- **When** executing the mutator call at **L6**, due to *call by value*, a copy of variable `i` is made.

  ⇒ The assignment `i = i + 1` is only effective on this copy, not the original variable `i` itself.

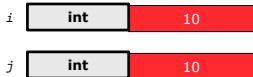- ∴ **After** the mutator call at **L6**, variable `i` still stores 10.

Call by Value (2.2.2)

Before `reassignInt`	During `reassignInt`	After `reassignInt`
	i `int` 10	*i* `int` 10
i `int` 10	*j* `int` 10	*j* `int` 11

Call by Value (2.3.1)

```java
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```
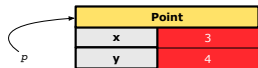
```java
1   @Test
2   public void testCallByRef_1() {
3     Util u = new Util();
4     Point p = new Point(3, 4);
5     Point refOfPBefore = p;
6     u.reassignRef(p);
7     assertTrue(p == refOfPBefore);
8     assertTrue(p.getX() == 3);
9     assertTrue(p.getY() == 4);
10  }
```
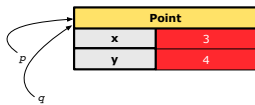
- **Before** the mutator call at **L6**, *reference* variable p stores the *address* of some Point object (whose x is 3 and y is 4).

- **When** executing the mutator call at **L6**, due to *call by value*, a *copy of address* stored in p is made.
  - ⇒ The assignment p = np is only effective on this copy, not the original variable p itself.

- ∴ **After** the mutator call at **L6**, variable p still stores the original address (i.e., same as refOfPBefore).
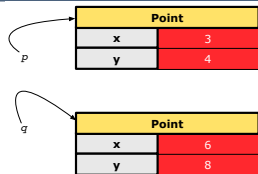
Call by Value (2.3.2)

Before `reassignRef`	During `reassignRef`	After `reassignRef`

Call by Value (2.4.1)

```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```
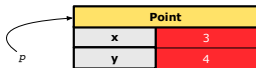
```
1   @Test
2   public void testCallByRef_2() {
3     Util u = new Util();
4     Point p = new Point(3, 4);
5     Point refOfPBefore = p;
6     u.changeViaRef(p);
7     assertTrue(p == refOfPBefore);
8     assertTrue(p.getX() == 6);
9     assertTrue(p.getY() == 8);
10  }
```
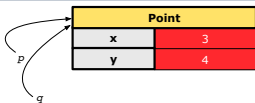
- *Before* the mutator call at **L6**, reference variable p stores the address of some Point object (whose x is 3 and y is 4).

- **When** executing the mutator call at **L6**, due to call by value, a copy of address stored in p is made. [**Alias**: p and q store same address.]

  ⇒ q.moveHorizontally impacts the *same object* referenced by p and q.

- ∴ *After* the mutator call at **L6**, variable p still stores the original address (i.e., same as refOfPBefore), but its x and y values have been modified via q.
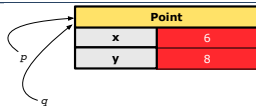
Before `changeViaRef` || During `changeViaRef` || After `changeViaRef`

Aggregation vs. Composition: Terminology

Container object: an object that contains others.

Containee object: an object that is contained within another.

- e.g., Each course has a faculty member as its instructor.
  - *Container*: Course   *Containee*: Faculty.
- e.g., Each student is registered in a list of courses; Each faculty member teaches a list of courses.
  - *Container*: Student, Faculty   *Containees*: Course.

    e.g., eecs2030 taken by jim (student) and taught by tom (faculty).

    ⇒ *Containees* **may** be *shared* by different instances of *containers*.

    e.g., When EECS2030 is finished, jim and jackie still exist!

    ⇒ *Containees* **may** *exist independently* without their *containers*.
- e.g., In a file system, each directory contains a list of files.
  - *Container*: Directory   *Containees*: File.

    e.g., Each file has exactly one parent directory.

    ⇒ A *containee* **may** be *owned* by only one *container*.

    e.g., Deleting a directory also deletes the files it contains.

    ⇒ *Containees* **may** *co-exist* with their *containers*.

Aggregation: Independent Containees Shared by Containers (1.1)

```
Course  ◇————————— prof ————— Faculty
                        1
```
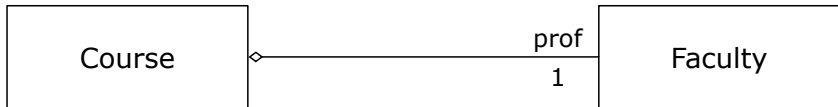
```java
public class Course {
  private String title;
  private Faculty prof;
  public Course(String title) {
    this.title = title;
  }
  public void setProf(Faculty prof) {
    this.prof = prof;
  }
  public Faculty getProf() {
    return this.prof;
  }
}
```
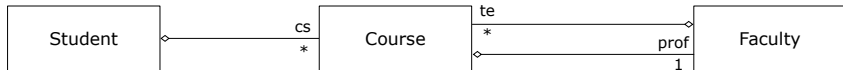
```java
public class Faculty {
  private String name;
  public Faculty(String name) {
    this.name = name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return this.name;
  }
}
```

Aggregation: Independent Containees Shared by Containers (1.2)

```java
@Test
public void testAggregation1() {
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  Faculty prof = new Faculty("Jackie");
  eecs2030.setProf(prof);
  eecs3311.setProf(prof);
  assertTrue(eecs2030.getProf() == eecs3311.getProf());
  /* aliasing */
  prof.setName("Jeff");
  assertTrue(eecs2030.getProf() == eecs3311.getProf());
  assertTrue(eecs2030.getProf().getName().equals("Jeff"));

  Faculty prof2 = new Faculty("Jonathan");
  eecs3311.setProf(prof2);
  assertTrue(eecs2030.getProf() != eecs3311.getProf());
  assertTrue(eecs2030.getProf().getName().equals("Jeff"));
  assertTrue(eecs3311.getProf().getName().equals("Jonathan"));
}
```

Aggregation: Independent Containees Shared by Containers (2.1)

```
Student  ◇━━━━━━ cs ━━━━━━  Course  ━━━━━━ te ━━━━━━  Faculty
                      *                    *      prof
                                                  1
```

```java
public class Student {
  private String id; Course[] cs; int noc; /* # of courses */
  public Student(String id) { ... }
  public void addCourse(Course c) { ... }
  public Course[] getCS() { ... }
}
```

```java
public class Course { private String title; private Faculty prof; }
```

```java
public class Faculty {
  private String name; Course[] te; int not; /* # of teaching */
  public Faculty(String name) { ... }
  public void addTeaching(Course c) { ... }
  public Course[] getTE() { ... }
}
```

Aggregation: Independent Containees Shared by Containers (2.2)

```
@Test
public void testAggregation2() {
  Faculty p = new Faculty("Jackie");
  Student s = new Student("Jim");
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  eecs2030.setProf(p);
  eecs3311.setProf(p);
  p.addTeaching(eecs2030);
  p.addTeaching(eecs3311);
  s.addCourse(eecs2030);
  s.addCourse(eecs3311);

  assertTrue(eecs2030.getProf() == s.getCS()[0].getProf());
  assertTrue(s.getCS()[0].getProf()
             == s.getCS()[1].getProf());
  assertTrue(eecs3311 == s.getCS()[1]);
  assertTrue(s.getCS()[1] == p.getTE()[1]);
}
```

The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.
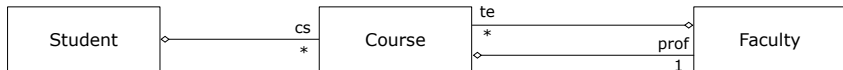

```
public class Student {
  private String id;
  private Course[] cs;
}
```

```
public class Course {
  private String title;
  private Faculty prof;
}
```

```
public class Faculty {
  private String name;
  private Course[] te;
}
```

- <u>Assume</u>: *private* attributes and *public* accessors
- *Aggregation links* between classes constrain how you can *navigate* among these classes.
- In the context of class Student:
  - Writing *cs* denotes the array of registered courses.
  - Writing *cs[i]* (where i is a valid index) navigates to the class Course, which changes the context to class Course.

OOP: The Dot Notation (3.2)

```
public class Student {
  private String id;
  private Course[] cs;
}
```

```
public class Course {
  private String title;
  private Faculty prof;
}
```

```
public class Faculty {
  private String name;
  private Course[] te;
}
```

```
public class Student {
  ... /* attributes */
  /* Get the student's id */
  public String getID() { return this.id; }
  /* Get the title of the ith course */
  public String getTitle(int i) {
    return this.cs[i].getTitle();
  }
  /* Get the instructor's name of the ith course */
  public String getName(int i) {
    return this.cs[i].getProf.getName();
  }
}
```

```
public class Student {
  private String id;
  private Course[] cs;
}
```

```
public class Course {
  private String title;
  private Faculty prof;
}
```

```
public class Faculty {
  private String name;
  private Course[] te;
}
```

```
public class Course {
  ... /* attributes */
  /* Get the course's title */
  public String getTitle() { return this.title; }
  /* Get the instructor's name */
  public String getName() {
    return this.prof.getName();
  }
  /* Get title of ith teaching course of the instructor */
  public String getTitle(int i) {
    return this.prof.getTE()[i].getTitle();
  }
}
```

OOP: The Dot Notation (3.4)

```
public class Student {
  private String id;
  private Course[] cs;
}
```

```
public class Course {
  private String title;
  private Faculty prof;
}
```

```
public class Faculty {
  private String name;
  private Course[] te;
}
```

```
public class Faculty {
  ... /* attributes */
  /* Get the instructor's name */
  public String getName() {
    return this.name;
  }
  /* Get the title of ith teaching course */
  public String getTitle(int i) {
    return this.te[i].getTitle();
  }
}
```

Directory ◆——————— File

parent | 1 .... files | *
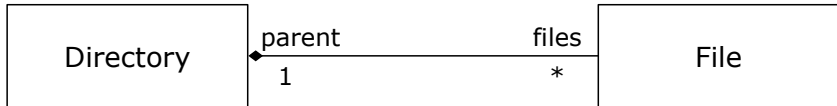
Requirement: Files are not shared among directories.

Assume: *private* attributes and *public* accessors
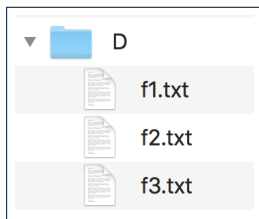
```
class File {
  String name;
  File(String name) {
    this.name = name;
  }
}
```

```
class Directory {
  String name;
  File[] files;
  int nof; /* num of files */
  Directory(String name) {
    this.name = name;
    files = new File[100];
  }
  void addFile(String fileName) {
    files[nof] = new File(fileName);
    nof ++;
  }
}
```

Composition: Dependent Containees Owned by Containers (1.2.1)



```
1  @Test
2  public void testComposition() {
3    Directory d1 = new Directory("D");
4    d1.addFile("f1.txt");
5    d1.addFile("f2.txt");
6    d1.addFile("f3.txt");
7    assertTrue(d1.getFiles()[0].getName().equals("f1.txt"));
8  }
```

- **L4**: 1st `File` object is created and *owned exclusively* by d1.
  No other directories are sharing this `File` object with d1.
- **L5**: 2nd `File` object is created and *owned exclusively* by d1.
  No other directories are sharing this `File` object with d1.
- **L6**: 3rd `File` object is created and *owned exclusively* by d1.
  No other directories are sharing this `File` object with d1.

Composition: Dependent Containees Owned by Containers (1.2.2)

Right before test method `testComposition` terminates:
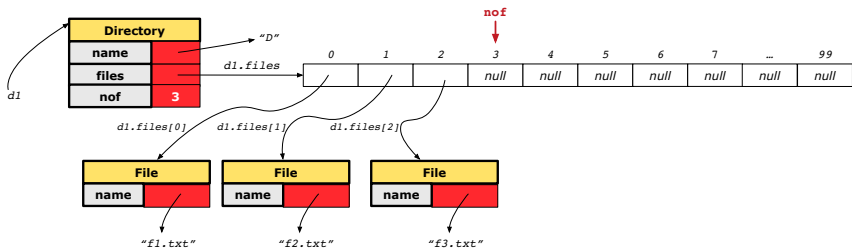
Composition: Dependent Containees Owned by Containers (1.3)

Problem: Implement a <mark>*copy constructor*</mark> for `Directory`.

A ***copy constructor*** is a constructor which initializes attributes from the argument object `other` (of the ***same type*** `Directory`).

```
class Directory {
 Directory(Directory other) {
   /* Initialize attributes via attributes of 'other'. */
 }
}
```

Hints:
- The implementation should be consistent with the effect of copying and pasting a directory.
- Separate copies of files are created.

Composition: Dependent Containees Owned by Containers (1.4.1)

Version 1: *Shallow Copy* by copying all attributes using =.

```
class Directory {
 Directory(Directory other) {
   /* value copying for primitive type */
   nof = other.nof;
   /* address copying for reference type */
   name = other.name; files = other.files; } }
```

Is a shallow copy satisfactory to support composition?
i.e., Does it still forbid sharing to occur?   [*NO*]
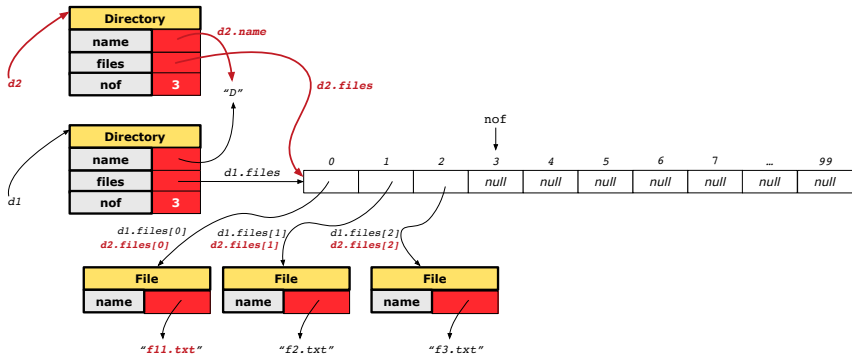
```
@Test
public void testShallowCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.getFiles() == d2.getFiles()); /* violation of composition */
  d2.getFiles()[0].changeName("f11.txt");
  assertFalse(d1.getFiles()[0].getName().equals("f1.txt"));
}
```

Composition: Dependent Containees Owned by Containers (1.4.2)

Right before test method `testShallowCopyConstructor` terminates:

Composition: Dependent Containees Owned by Containers (1.5.1)

Version 2: a *Deep Copy*

```java
class File {
  File(File other) {
    this.name =
      new String(other.name);
  }
}
```

```java
class Directory {
  Directory(String name) {
    this.name = new String(name);
    files = new File[100]; }
  Directory(Directory other) {
    this (other.name);
    for(int i = 0; i < other.nof; i ++) {
      File src = other.files[i];
      File nf = new File(src);
      this.addFile(nf);
    }
  }
  void addFile(File f) { ... }
}
```

```java
@Test
public void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.getFiles() != d2.getFiles()); /* composition preserved */
  d2.getFiles()[0].changeName("f11.txt");
  assertTrue(d1.getFiles()[0].getName().equals("f1.txt"));
}
```

Composition: Dependent Containees Owned by Containers (1.5.2)

Right before test method `testDeepCopyConstructor` terminates:

Composition: Dependent Containees Owned by Containers (1.5.3)

Q: Composition Violated?
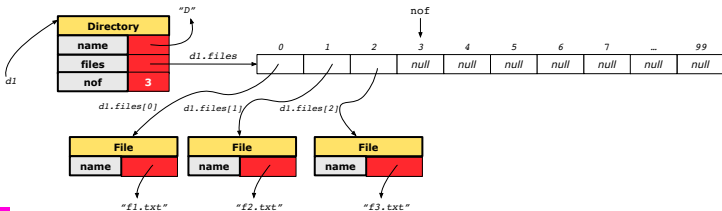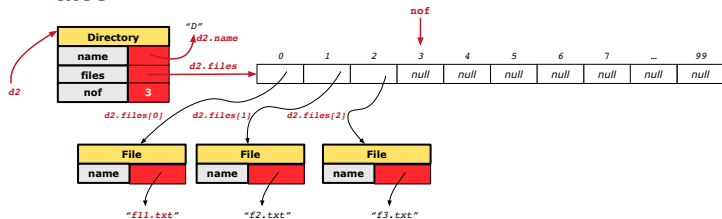
```
class File {
  File(File other) {
    this.name =
      new String(other.name);
  }
}
```

```
class Directory {
  Directory(String name) {
    this.name = new String(name);
    files = new File[100]; }
  Directory(Directory other) {
    this (other.name);
    for(int i = 0; i < other.nof; i ++) {
      File src = other.files[i];
      this.addFile(src);
    }
  }
  void addFile(File f) { ... }
}
```

```
@Test
public void testDeepCopyConstructor() {
  Directory d1 = new Directory("D");
  d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
  Directory d2 = new Directory(d1);
  assertTrue(d1.getFiles() != d2.getFiles()); /* composition preserved */
  d2.getFiles()[0].changeName("f11.txt");
  assertTrue(d1.getFiles()[0] == d2.getFiles()[0]); /* composition violated! */
}
```

Composition: Dependent Containees Owned by Containers (1.6)

Exercise: Implement the accessor in class `Directory`

```
class Directory {
  File[] files;
  int nof;
  File[] getFiles() {
    /* Your Task */
  }
}
```

so that it *preserves composition*, i.e., does not allow references of files to be shared.

Aggregation vs. Composition (1)

Terminology:
- **Container** object: an object that contains others.
- **Containee** object: an object that is contained within another.

Aggregation :
- Containees (e.g., `Course`) may be *shared* among containers (e.g., `Student`, `Faculty`).
- Containees *exist independently* without their containers.
- When a container is destroyed, its containees still exist.

Composition :
- Containers (e.g, `Directory`, `Department`) *own* exclusive access to their containees (e.g., `File`, `Faculty`).
- Containees cannot exist without their containers.
- Destroying a container destroys its containeees *cascadingly* .

Aggregation vs. Composition (2)

Aggregations and *Compositions* may exist at the same time!
e.g., Consider a workstation:

- Each workstation owns CPU, monitor, keyword.   [**compositions**]
- All workstations share the same network.   [**aggregations**]
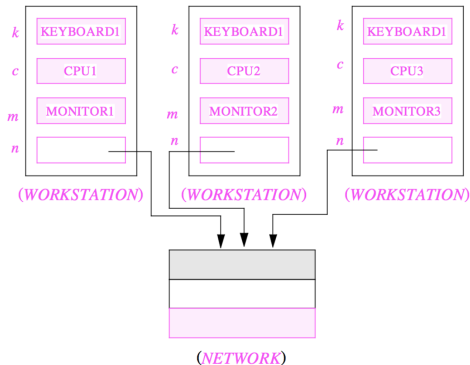
Aggregation vs. Composition (3)

Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book be implemented as an *aggregation* or a *composition*?

author as an *aggregation*	author as a *composition*
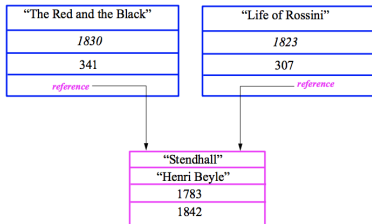
Hyperlinked author page	Physical printed copies
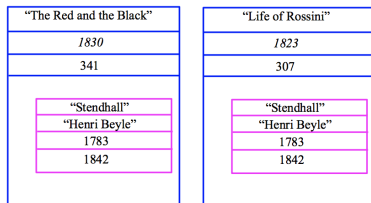
Reproduce the *aggregation* and *composition* code examples
in Eclipse.

Tip. Use the debugger to verify whether or not there is *sharing*.

Index (1)

Index (2)

Index (4)

Inheritance

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

- <u>Alternative</u> designs to **inheritance**
- Using **inheritance** for <u>code reuse</u>
- ***Static Types***, <u>Expectations</u>, ***Dynamic Types***
- *Polymorphism*
  (variable assignments, method arguments & return values)
- *Dynamic Binding*
- ***Type Casting***

Why Inheritance: A Motivating Example

Problem: A student management system stores data about
students. There are two kinds of university students: resident
students and non-resident students. Both kinds of students
have a name and a list of registered courses. Both kinds of
students are restricted to register for no more than 10 courses.
When calculating the tuition for a student, a base amount is first
determined from the list of courses they are currently registered
(each course has an associated fee). For a non-resident
student, there is a discount rate applied to the base amount to
waive the fee for on-campus accommodation. For a resident
student, there is a premium rate applied to the base amount to
account for the fee for on-campus accommodation and meals.
Tasks: Write Java classes that satisfy the above problem
statement. At runtime, each type of student must be able to
register a course and calculate their tuition fee.

Why Inheritance: A Motivating Example

Problem: A *student management system* stores data about students. There are two kinds of university students: *resident* students and *non-resident* students. Both kinds of students have a *name* and a list of *registered courses*. Both kinds of students are restricted to *register* for no more than 10 courses. When *calculating the tuition* for a student, a base amount is first determined from the list of courses they are currently registered (each course has an associated fee). For a non-resident student, there is a *discount rate* applied to the base amount to waive the fee for on-campus accommodation. For a resident student, there is a *premium rate* applied to the base amount to account for the fee for on-campus accommodation and meals.
Tasks: Write Java classes that satisfy the above problem statement. At runtime, each type of student must be able to register a course and calculate their tuition fee.

No Inheritance: `ResidentStudent` Class

```java
public class ResidentStudent {
  private String name;
  private Course[] courses; private int noc;
  private double premiumRate; /* assume a mutator for this */
  public ResidentStudent (String name) {
    this.name = name;
    this.courses = new Course[10];
  }
  public void register(Course c) {
    this.courses[this.noc] = c;
    this.noc ++;
  }
  public double getTuition() {
    double tuition = 0;
    for(int i = 0; i < this.noc; i ++) {
      tuition += this.courses[i].fee;
    }
    return tuition * this.premiumRate;
  }
}
```

```java
public class NonResidentStudent {
 private String name;
 private Course[] courses; private int noc;
 private double discountRate; /* assume a mutator for this */
 public NonResidentStudent (String name) {
   this.name = name;
   this.courses = new Course[10];
 }
 public void register(Course c) {
   this.courses[this.noc] = c;
   this.noc ++;
 }
 public double getTuition() {
   double tuition = 0;
   for(int i = 0; i < this.noc; i ++) {
     tuition += this.courses[i].fee;
   }
   return tuition * this. discountRate ;
 }
}
```

No Inheritance: Testing Student Classes

```java
public class Course {
  private String title; private double fee;
  public Course(String title, double fee) {
    this.title = title; this.fee = fee;
  }
}
```

```java
public class StudentTester {
  public static void main(String[] args) {
    Course c1 = new Course("EECS2030", 500.00); /* title and fee */
    Course c2 = new Course("EECS3311", 500.00); /* title and fee */
    ResidentStudent jim = new ResidentStudent("J. Davis");
    jim.setPremiumRate(1.25);
    jim.register(c1); jim.register(c2);
    NonResidentStudent jeremy = new NonResidentStudent("J. Gibbons");
    jeremy.setDiscountRate(0.75);
    jeremy.register(c1); jeremy.register(c2);
    System.out.println("Jim pays " + jim.getTuition());
    System.out.println("Jeremy pays " + jeremy.getTuition());
  }
}
```

- Implementations for the two student classes seem to work.

  But can you see any potential problems with it?

  Hint. Maintenance of code

- The code of the two student classes share a lot in common.
  - *Duplicates of code make it hard to maintain your software!*
  - This means that when there is a change of policy on the common part, we need modify *more than one places*.
  - This violates the so-called *single-choice design principle*.

What if the way for registering a course changes?

e.g.,

```
public void register(Course c) throws TooManyCoursesException {
  if (this.noc >= MAX_ALLOWANCE) {
    throw new TooManyCoursesException("Too many courses");
  }
  else {
    this.courses[this.noc] = c;
    this.noc ++;
  }
}
```

Changes needed for `register` method in **both** student classes!

What if the way for calculating the base tuition changes?

e.g.,

```
public double getTuition() {
  double tuition = 0;
  for(int i = 0; i < this.noc; i ++) {
    tuition += this.courses[i].fee;
  }
  /* ... can be premiumRate or discountRate */
  return tuition * inflationRate * ...;
}
```

Changes needed for getTuition method in *both* student classes!

No Inheritance:
A Collection of Various Kinds of Students

How can we define a class `StudentManagementSystem` that
contains a list of *resident* and *non-resident* students?

```
public class StudentManagementSystem {
 private ResidentStudent[] rss;
 private NonResidentStudent[] nrss;
 private int nors; /* number of resident students */
 private int nonrs; /* number of non-resident students */
 public void addRS(ResidentStudent rs){ rss[nors]=rs; nors++; }
 public void addNRS(NonResidentStudent nrs){ nrss[nonrs]=nrs;nonrs++; }
 public void registerAll(Course c) {
   for(int i = 0; i < nors; i ++) { rss[i].register(c); }
   for(int i = 0; i < nonrs; i ++) { nrss[i].register(c); }
 }
}
```

But what if we later on introduce ***more kinds of students***?

Very ***inconvenient*** to handle each list of students ***separately***!

a polymorphic collection of students

CollectionOfStuffs

animal
Cat
Dog

furniture
Chair
Desk

shape
Circle
Square
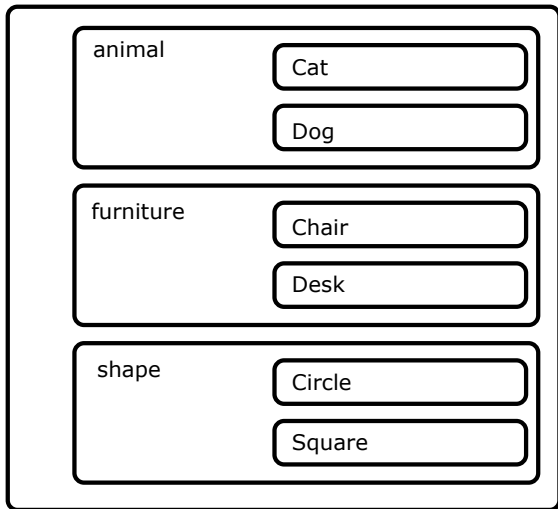
Visibility of Classes

- Only <u>one</u> modifier for declaring visibility of classes: ***public***.
- Use of ***private*** is forbidden for declaring a class.

    e.g., | ***private* `class` Chair** | is **not** allowed!!

- **Visibility** of <u>a class</u> may be declared using a <u>modifier</u>, indicating that it is accessible:
    1. Across <u>classes</u> within its residing package   [no modifier]
        e.g., Declare | **class** Chair { … } |
    2. Across <u>packages</u>   [***public***]
        e.g., Declare | ***public* class** Chair { … } |
- Consider class `Chair` which resides in:
    ○ package `furniture`
    ○ project `CollectionOfStuffs`

CollectionOfStuffs

animal

Cat

Dog

furniture

public class Chair

Desk

shape

Circle

Square
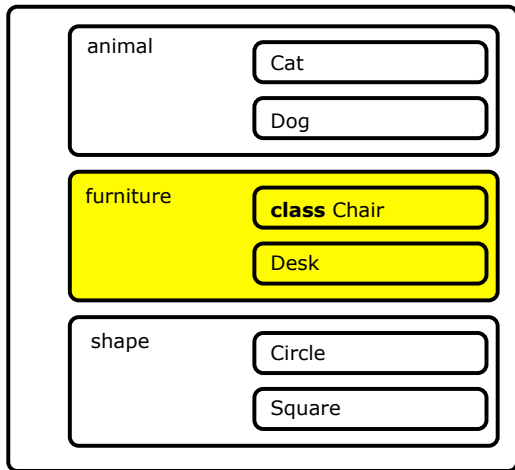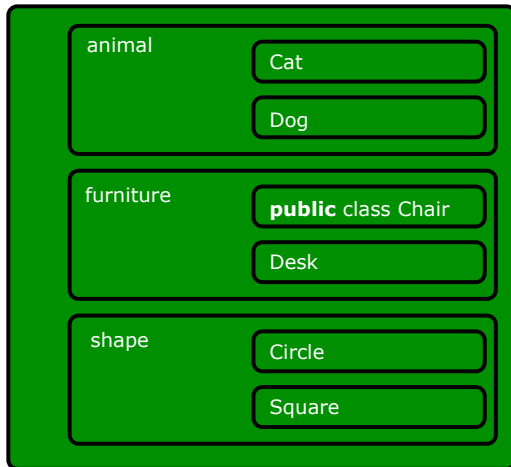
Visibility of Attributes/Methods: Using Modifiers to Define Scopes

- Two modifiers for declaring visibility of attributes/methods: *public* and *private*
- **Visibility** of <u>an attribute or a method</u> may be declared using a <u>modifier</u>, indicating that it is accessible:

  1. Within its residing <u>class</u> (***most*** restrictive)   [*private*]

    e.g., Declare attribute $\boxed{\textit{private} \text{ int i;}}$

    e.g., Declare method $\boxed{\textit{private} \text{ void m()\{\};}}$

  2. Across <u>classes</u> within its residing package   [no modifier]

    e.g., Declare attribute $\boxed{\text{int i;}}$

    e.g., Declare method $\boxed{\text{void m()\{\};}}$

  3. Across <u>packages</u> (***least*** restrictive)   [*public*]

    e.g., Declare attribute $\boxed{\textit{public} \text{ int i;}}$

    e.g., Declare method $\boxed{\textit{public} \text{ void m()\{\};}}$

- Consider attributes `i` and `m` residing in:

  Class `Chair`; Package `furniture`; Project `CollectionOfStuffs`.

CollectionOfStuffs

animal
- Cat
- Dog

furniture
- Chair   **private** i, m
- Desk

shape
- Circle
- Square

CollectionOfStuffs

animal
- Cat
- Dog

furniture
- Chair **public** i, m
- Desk

shape
- Circle
- Square

Use of the `protected` Modifier

- ***private*** attributes are not inherited to subclasses.
- package-level attributes (i.e., with **no modifier**) and project-level attributes (i.e., ***public***) are inherited.
- What if we want attributes to be:
  - *visible* to sub-classes <u>outside</u> the current package, but still
  - *invisible* to other <u>non-</u>sub-classes <u>outside</u> the current package?

  Use ***protected***!

Visibility of Attr./Meth.: Across All Methods Same Package and Sub-Classes (`protected`)
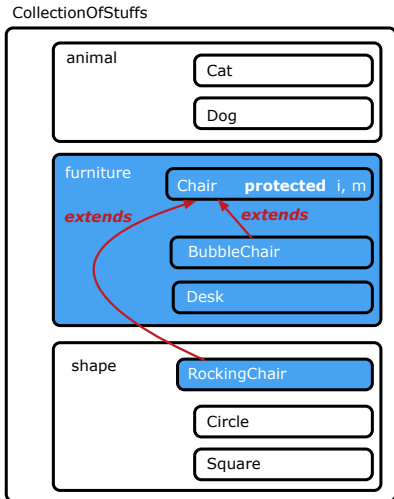
Visibility of Attributes/Methods

modifier \ scope	CLASS	PACKAGE	SUBCLASS (same pkg)	SUBCLASS (different pkg)	NON-SUBCLASS (across Project)
public					
protected					
no modifier					
private					

For the rest of this lecture, for simplicity, we assume that:

All relevant parent/child classes are in the same package .

⇒ Attributes with **no modifiers** (***package**-level visibility) suffice.

⇒ Methods with **no modifiers** (***package**-level visibility) suffice.

Inheritance Architecture
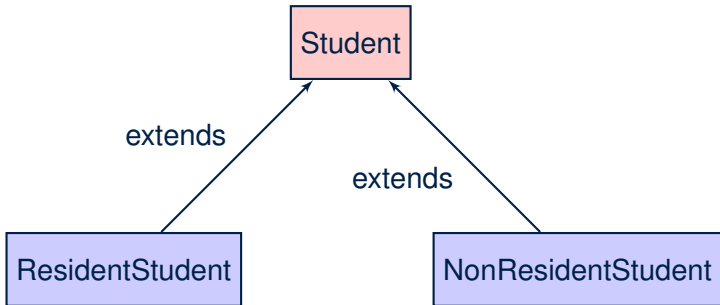
```java
class Student {
 String name;
 Course[] courses; int noc;
 Student (String name) {
   this.name = name;
   this.courses = new Course[10];
 }
 void register(Course c) {
   this.courses[this.noc] = c;
   this.noc ++;
 }
 double getTuition() {
   double tuition = 0;
   for(int i = 0; i < this.noc; i ++) {
     tuition += this.courses[i].fee;
   }
   return tuition; /* base amount only */
 }
}
```

Inheritance:
The **ResidentStudent** Child/Sub Class

```
1  class ResidentStudent  extends Student {
2    double premiumRate; /* there's a mutator method for this */
3    ResidentStudent (String name) { super(name); }
4  /* register method is inherited */
5  double getTuition() {
6    double base = super.getTuition();
7    return base * premiumRate ;
8  }
9  }
```

- **L1** declares that ResidentStudent inherits all attributes and methods (except constructors) from Student.
- There is no need to repeat the register method
- Use of *super* in **L4** is as if calling Student(name)
- Use of *super* in **L8** returns what getTuition() in Student returns.
- Use *super* to refer to attributes/methods defined in the super class:
  | **super**.*name* | | **super**.*register*(*c*) |

Inheritance:
The `NonResidentStudent` Child/Sub Class

```
1  class NonResidentStudent  extends Student {
2    double discountRate;  /* there's a mutator method for this */
3    NonResidentStudent (String name) { super(name); }
4  /* register method is inherited */
5    double getTuition() {
6      double base = super.getTuition();
7      return base * discountRate ;
8    }
9  }
```

- **L1** declares that NonResidentStudent inherits all attributes and methods (except constructors) from Student.
- There is no need to repeat the register method
- Use of *super* in **L4** is as if calling Student(name)
- Use of *super* in **L8** returns what getTuition() in Student returns.
- Use *super* to refer to attributes/methods defined in the super class:
  `super.name` , `super.register(c)` .

- The class that defines the common attributes and methods is called the *parent* or *super* class.

- Each "extended" class is called a *child* or *sub* class.

Using Inheritance for Code Reuse

Inheritance in Java allows you to:

○ Define **common attributes and methods** in a separate class.
  e.g., the `Student` class

○ Define an "extended" version of the class which:

  • **inherits** definitions of all attributes and methods
    e.g., `name`, `courses`, `noc`
    e.g., `register`
    e.g., base amount calculation in `getTuition`
    This means code reuse and elimination of code duplicates!

  • **defines** new attributes and methods if necessary
    e.g., `setPremiumRate` for `ResidentStudent`
    e.g., `setDiscountRate` for `NonResidentStudent`

  • **redefines/overrides** methods if necessary
    e.g., compounded tuition for `ResidentStudent`
    e.g., discounted tuition for `NonResidentStudent`

- A child class inherits **all** <u>non-private</u> attributes from its parent class.

  ⇒ A child instance has **at least as many** attributes as an instance of its parent class.

  Consider the following instantiations:

```
Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

- How will these initial objects look like?

LASSONDE
SCHOOL OF ENGINEERING

s

Student	
name	
numberOfCourses	0
registeredCourses	

"Stella"

	0	1		8	9
	null	null	...	null	null

rs

ResidentStudent	
name	
numberOfCourses	0
registeredCourses	
premiumRate	

"Rachael"

	0	1		8	9
	null	null	...	null	null

nrs

NonResidentStudent	
name	
numberOfCourses	0
registeredCourses	
discountRate	

"Nancy"

	0	1		8	9
	null	null	...	null	null
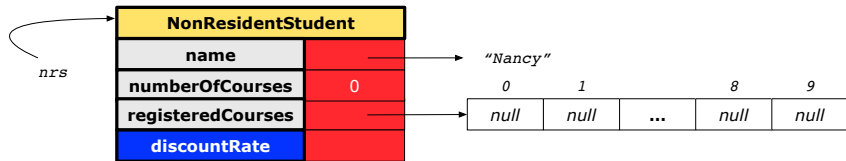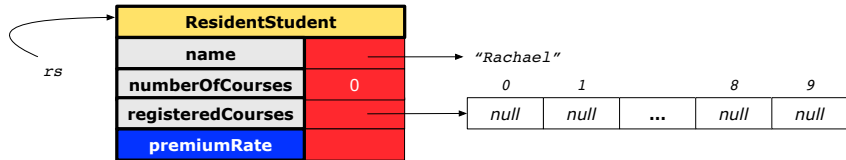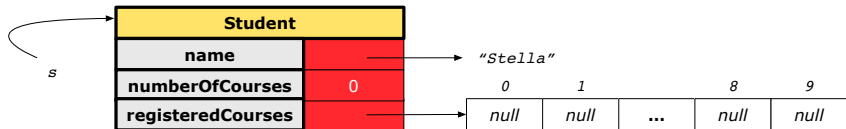
Testing the Two Student Sub-Classes

```java
public class StudentTester {
 public static void main(String[] args) {
   Course c1 = new Course("EECS2030", 500.00); /* title and fee */
   Course c2 = new Course("EECS3311", 500.00); /* title and fee */
   ResidentStudent jim = new ResidentStudent("J. Davis");
   jim.setPremiumRate(1.25);
   jim.register(c1); jim.register(c2);
   NonResidentStudent jeremy = new NonResidentStudent("J. Gibbons");
   jeremy.setDiscountRate(0.75);
   jeremy.register(c1); jeremy.register(c2);
   System.out.println("Jim pays " + jim.getTuition());
   System.out.println("Jeremy pays " + jeremy.getTuition());
 }
}
```

- The software can be used in the exact same way as before (because we did not modify *method headers*).
- But now the internal structure of code has been made *maintainable* using *inheritance*.

Inheritance Architecture: Static Types & Expectations

Student(String name)
void register(Course c)
double getTuition()

```
           Student
```

String name
Course[] courses /* registered courses (rcs) */
int noc /* number of courses */

/ new attributes, new methods */*
ResidentStudent(String name)
double premiumRate
void setPremiumRate(double r)
/ redefined/overridden methods */*
double getTuition()

```
     ResidentStudent          NonResidentStudent
```

/ new attributes, new methods */*
NonResidentStudent(String name)
double discountRate
void setDiscountRate(double r)
/ redefined/overridden methods */*
double getTuition()

```
Student s = new Student("Stella");
ResidentStudent rs = new ResidentStudent("Rachael");
NonResidentStudent nrs = new NonResidentStudent("Nancy");
```

	name	rcs	noc	reg	getT	pr	setPR	dr	setDR
s.		✓				✗			
rs.		✓				✓		✗	
nrs.		✓				✗		✓	

Polymorphism: Intuition (1)

```
1  Student s = new Student("Stella");
2  ResidentStudent rs = new ResidentStudent("Rachael");
3  rs.setPremiumRate(1.25);
4  s = rs; /* Is this valid? */
5  rs = s; /* Is this valid? */
```

- Which one of **L4** and **L5** is *valid*? Which one is *invalid*?
- **Hints**:
  - **L1**: What *kind* of address can *s* store?   [Student]
    ∴ The context object *s* is *expected* to be used as:
    - *s*.register(eecs2030) and s.getTuition()
  - **L2**: What *kind* of address can *rs* store? [ResidentStudent]
    ∴ The context object *rs* is *expected* to be used as:
    - *rs*.register(eecs2030) and *rs*.getTuition()
    - *rs*.setPremiumRate(1.50)   [increase premium rate]

Polymorphism: Intuition (2)

```
1  Student s = new Student("Stella");
2  ResidentStudent rs = new ResidentStudent("Rachael");
3  rs.setPremiumRate(1.25);
4  s = rs; /* Is this valid? */
5  rs = s; /* Is this valid? */
```

- **rs** = **s** (**L5**) should be *invalid*:

Student	
name	"Stella"
registeredCourses	
numberOfCourses	0

0	1	2	...	28	29
null	null	null		null	null

ResidentStudent	
name	"Rachael"
registeredCourses	
numberOfCourses	0
premiumRate	1.25

0	1	2	...	28	29
null	null	null		null	null

- Since **rs** is declared of type ResidentStudent, a subsequent call **rs.**`setPremiumRate(1.50)` can be expected.
- **rs** is now pointing to a Student object.
- Then, what would happen to **rs.**`setPremiumRate(1.50)`?

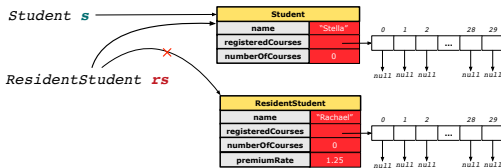 **CRASH** ∵ **rs**.premiumRate is *undefined*!!

Polymorphism: Intuition (3)

```
1  Student s = new Student("Stella");
2  ResidentStudent rs = new ResidentStudent("Rachael");
3  rs.setPremiumRate(1.25);
4  s = rs; /* Is this valid? */
5  rs = s; /* Is this valid? */
```

- **s** = **rs** (**L4**) should be *valid*:
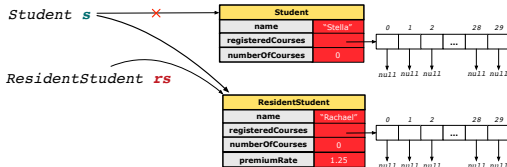
- Since **s** is declared of type Student, a subsequent call
  s.`setPremiumRate(1.50)` is *never* expected.
- **s** is now pointing to a ResidentStudent object.
- Then, what would happen to **s.**`getTuition()`?

  > OK   ∵ **s.**premiumRate is *never directly used*!!

Dynamic Binding: Intuition (1)

```
1  Course eecs2030 = new Course("EECS2030", 100.0);
2  Student s;
3  ResidentStudent rs = new ResidentStudent("Rachael");
4  NonResidentStudent nrs = new NonResidentStudent("Nancy");
5  rs.setPremiumRate(1.25); rs.register(eecs2030);
6  nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7  s = rs;  System.out.println(s.getTuition()); /* 125.0 */
8  s = nrs; System.out.println(s.getTuition()); /* 75.0 */
```

After `s = rs` (**L7**), s points to a `ResidentStudent` object.
⇒ Calling `s`.getTuition() applies the `premiumRate`.
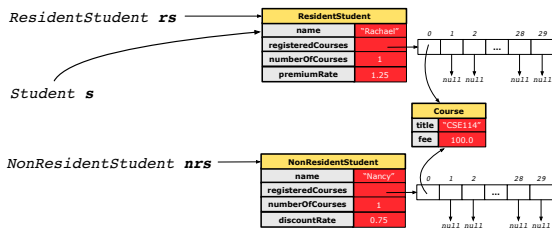
Dynamic Binding: Intuition (2)

```
1  Course eecs2030 = new Course("EECS2030", 100.0);
2  Student s;
3  ResidentStudent rs = new ResidentStudent("Rachael");
4  NonResidentStudent nrs = new NonResidentStudent("Nancy");
5  rs.setPremiumRate(1.25); rs.register(eecs2030);
6  nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7  s = rs;  System.out.println( s .getTuition()); /* 125.0 */
8  s = nrs;  System.out.println( s .getTuition()); /* 75.0 */
```

After `s = nrs` (**L8**), s points to a `NonResidentStudent` object.
⇒ Calling `s`.getTuition() applies the `discountRate`.

Multi-Level Inheritance Architecture

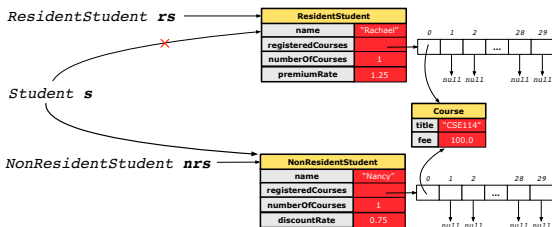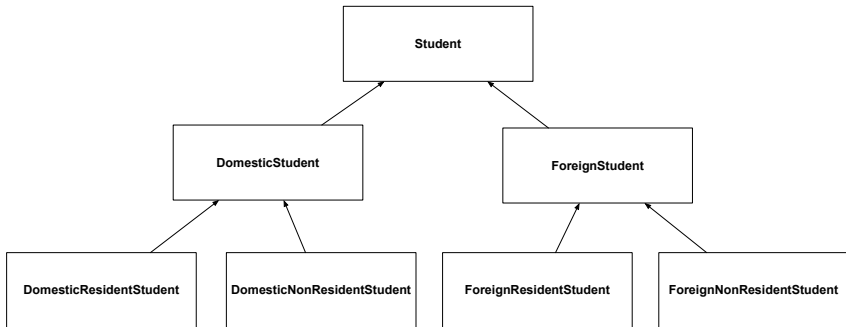# Multi-Level Inheritance Hierarchy: Smart Phones
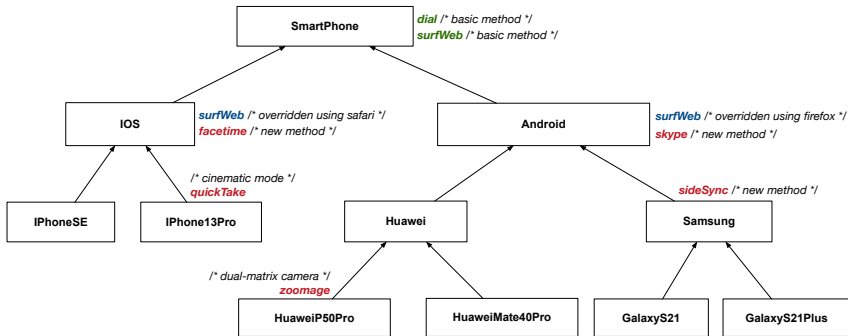
Inheritance Forms a Type Hierarchy

- A (data) *type* denotes a set of related ***runtime values***.
  - Every ***class*** can be used as a type: the set of runtime ***objects***.
- Use of ***inheritance*** creates a *hierarchy* of classes:
  - (Implicit) Root of the hierarchy is `Object`.
  - Each `extends` declaration corresponds to an upward arrow.
  - The `extends` relationship is ***transitive***: when A extends B and B extends C, we say A *indirectly* extends C.
    e.g., Every class implicitly `extends` the `Object` class.
- ***Ancestor*** vs. ***Descendant*** classes:
  - The *ancestor classes* of a class A are: A itself and all classes that A directly, or indirectly, extends.
    - A <u>inherits</u> all code (attributes and methods) from its ***ancestor classes***.
      ∴ A's instances have a ***wider range of expected usages*** (i.e., attributes and methods) than instances of its ***ancestor*** classes.
  - The *descendant classes* of a class A are: A itself and all classes that directly, or indirectly, extends A.
    - Code defined in A is <u>inherited to</u> all its ***descendant classes***.

Inheritance Accumulates Code for Reuse

- The **lower** a class is in the type hierarchy, the **more code** it accumulates from its **ancestor classes**:
  - A **descendant class** inherits all code from its **ancestor classes**.
  - A **descendant class** may also:
    - Declare new attributes
    - Define new methods
    - *Redefine* / *Override* inherited methods
- Consequently:
  - When being used as **context objects**,
    instances of a class' **descendant classes** have a **wider range of expected usages** (i.e., attributes and methods).
  - Given a **reference variable**, expected to store the address of an object of a particular class, we may **substitute** it with (**re-assign** it to) an object of any of its **descendant classes**.
  - e.g., When expecting a `SmartPhone` object, we may substitute it with either a `IPhone13Pro` or a `Samsung` object.
  - **Justification**: A **descendant class** contains **at least as many** methods as defined in its **ancestor classes** (but not vice versa!).

Static Types Determine Expectations

- A reference variable's **static type** is what we declare it to be.
  - `Student jim` declares jim's ST as `Student`.
  - `SmartPhone myPhone` declares myPhone's ST as `SmartPhone`.
  - The **static type** of a reference variable **never changes**.

- For a reference variable *v*, its **static type** C defines the **expected usages of v as a context object**.

- A method call `v.m(...)` is **compilable** if **m** is defined in C.
  - e.g., After declaring `Student jim`, we
    - **may** call `register` and `getTuition` on jim
    - **may** *not* call `setPremiumRate` (specific to a resident student) or `setDiscountRate` (specific to a non-resident student) on jim
  - e.g., After declaring `SmartPhone myPhone`, we
    - **may** call `dial` and `surfWeb` on myPhone
    - **may** *not* call `facetime` (specific to an IOS phone) or `skype` (specific to an Android phone) on myPhone

Substitutions via Assignments

- By declaring `C1` v1, **reference variable** v1 will store the **address** of an object "of class C1" at runtime.
- By declaring `C2` v2, **reference variable** v2 will store the **address** of an object "of class C2" at runtime.
- Assignment $\boxed{v1 = v2}$ **copies address** stored in v2 into v1.
  - v1 will instead point to wherever v2 is pointing to.   [*object alias*]
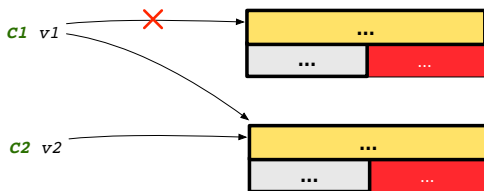
- In such assignment v1 = v2, we say that we *substitute* an object of (**static**) type C1 <u>by</u> an object of (**static**) type C2.
- *Substitutions* are subject to **rules**!

Rules of Substitution

When expecting an object of *static type* A:

- It is *safe* to substitute it with an object whose *static type* is any of the *descendant class* of A (including A).
  - ∵ Each *descendant class* of A, being the new substitute, is guaranteed to contain all (non-private) attributes/methods defined in A.
  - e.g., When expecting an IOS phone, you *can* substitute it with either an IPhoneSE or IPhone13Pro.
- It is *unsafe* to substitute it with an object whose *static type* is any of the *ancestor classes of A's parent* (excluding A).
  - ∵ Class A may have defined new methods that do not exist in any of its *parent's ancestor classes*.
  - e.g., When expecting IOS phone, *unsafe* to substitute it with a SmartPhone ∵ facetime not supported in Android phone.
- It is also *unsafe* to substitute it with an object whose *static type* is <u>neither</u> an ancestor <u>nor</u> a descendant of A.
  - e.g., When expecting IOS phone, *unsafe* to substitute it with a HuaweiP50Pro ∵ facetime not supported in Android phone.

Reference Variable: Dynamic Type

A **reference variable**'s *dynamic type* is the type of object that it is currently pointing to at <u>runtime</u>.

- The **dynamic type** of a reference variable *may change* whenever we *re-assign* that variable to a different object.
- There are two ways to re-assigning a reference variable.

Visualizing Static Type vs. Dynamic Type
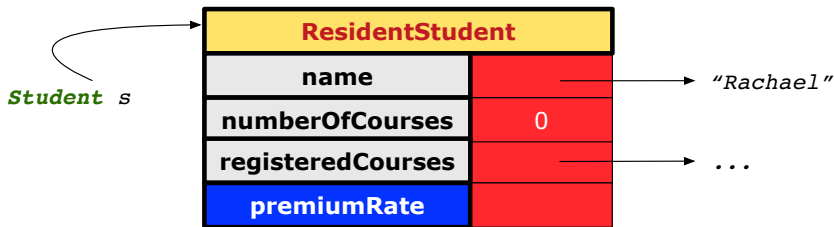
- Each segmented box denotes a *runtime* object.
- Arrow denotes a variable (e.g., `s`) storing the object's address.
  Usually, when the context is clear, we leave the variable's **static type** implicit (**Student**).
- Title of box indicates type of runtime object, which denotes the **dynamic type** of the variable (**ResidentStudent**).

Re-assigning a reference variable to a newly-created object:

○ **Substitution Principle** : the new object's class must be a
descendant class of the reference variable's **static type**.

○ e.g., `Student jim = new ResidentStudent(...)`
changes the **dynamic type** of jim to ResidentStudent.

○ e.g., `jim = new NonResidentStudent(...)`
changes the **dynamic type** of jim to NonResidentStudent.

○ e.g., `ResidentStudent jeremy = new Student(...)`
is illegal because Studnet is **not** a descendant class of the
static type of jeremy (i.e., ResidentStudent).

Reference Variable:
Changing Dynamic Type (2)

Re-assigning a reference variable v to an existing object that is referenced by another variable `other` (i.e., $\boxed{\texttt{v = other}}$):

- $\boxed{\textit{\textbf{Substitution Principle}}}$: the static type of `other` must be a $\boxed{\textit{descendant class}}$ of v's **static type**.
- e.g., Say we declare

```
Student jim = new Student(...);
ResidentStudent rs = new ResidentStudnet(...);
NonResidentStudnet nrs = new NonResidentStudent(...);
```

- $\boxed{\texttt{jim = rs}}$ ✓
  changes the *dynamic type* of `jim` to the dynamic type of `rs`
- $\boxed{\texttt{jim = nrs}}$ ✓
  changes the *dynamic type* of `jim` to the dynamic type of `nrs`
- $\boxed{\texttt{rs = jim}}$ ✗
- $\boxed{\texttt{nrs = jim}}$ ✗

Polymorphism and Dynamic Binding (1)

- *Polymorphism* : An object variable may have *"multiple possible shapes"* (i.e., allowable *dynamic types*).
  - Consequently, there are *multiple possible versions* of each method that may be called.
    - e.g., A *Student* variable may have the *dynamic type* of **Student**, **ResidentStudent**, or **NonResidentStudent**,
    - This means that there are three possible versions of the `getTuition()` that may be called.
- *Dynamic binding* : When a method `m` is called on an object variable, the version of `m` corresponding to its *"current shape"* (i.e., one defined in the *dynamic type* of *m*) will be called.

```
Student jim = new ResidentStudent(...);
jim.getTuition(); /* version in ResidentStudent */
jim = new NonResidentStudent(...);
jim.getTuition(); /* version in NonResidentStudent */
```

Polymorphism and Dynamic Binding (2.1)

```
class Student {...}
class ResidentStudent extends Student {...}
class NonResidentStudent extends Student {...}
```

```
class StudentTester1 {
  public static void main(String[] args) {
    Student jim = new Student("J. Davis");
    ResidentStudent rs = new ResidentStudent("J. Davis");
    jim = rs;  /* legal */
    rs = jim;  /* illegal */

    NonResidentStudnet nrs = new NonResidentStudent("J. Davis");
    jim = nrs;  /* legal */
    nrs = jim;  /* illegal */
  }
}
```

Polymorphism and Dynamic Binding (2.2)

```
class Student {...}
class ResidentStudent extends Student {...}
class NonResidentStudent extends Student {...}
```

```
class StudentTester2 {
  public static void main(String[] args) {
    Course eecs2030 = new Course("EECS2030", 500.0);
    Student jim = new Student("J. Davis");
    ResidentStudent rs = new ResidentStudent("J. Davis");
    rs.setPremiumRate(1.5);
    jim = rs ;
    System.out.println( jim.getTuition() );  /* 750.0 */
    NonResidentStudnet nrs = new NonResidentStudent("J. Davis");
    nrs.setDiscountRate(0.5);
    jim = nrs ;
    System.out.println( jim.getTuition() );  /* 250.0 */
  }
}
```
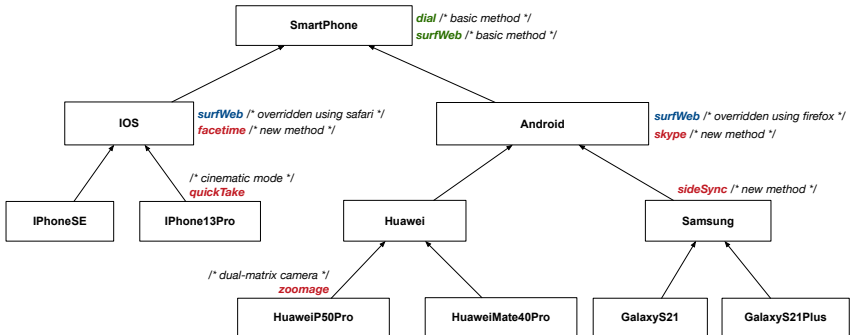
```
class SmartPhoneTest1 {
 public static void main(String[] args) {
   SmartPhone myPhone;
   IOS ip = new IPhoneSE();
   Samsung ss = new GalaxyS21Plus();
   myPhone = ip;  /* legal */
   myPhone = ss;  /* legal */

   IOS presentForHeeyeon;
   presentForHeeyeon = ip;  /* legal */
   presentForHeeyeon = ss;  /* illegal */
 }
}
```

Polymorphism and Dynamic Binding (3.3)

```
class SmartPhoneTest2 {
 public static void main(String[] args) {
   SmartPhone myPhone;
   IOS ip = new IPhone13Pro();
   myPhone = ip;
   myPhone. surfWeb ();  /* version of surfWeb in IPhone13Pro */

   Samsung ss = new GalaxyS21();
   myPhone = ss;
   myPhone. surfWeb ();  /* version of surfWeb in GalaxyS21 */
 }
}
```

Reference Type Casting: Motivation (1.1)

```
1   Student jim = new ResidentStudent("J. Davis");
2   ResidentStudent rs = jim;
3   rs.setPremiumRate(1.5);
```

- **L1** is *legal*: ResidentStudent is a **descendant class** of the *static type* of jim (i.e., Student).
- **L2** is *illegal*: jim's *ST* (i.e., Student) is **not** a **descendant class** of rs's *ST* (i.e., ResidentStudent).

  Java compiler is *unable to infer* that jim's *dynamic type* in **L2** is ResidentStudent!

- Force the Java compiler to believe so via a cast in **L2**:

  ```
  ResidentStudent rs = (ResidentStudent) jim;
  ```

  - The cast (*ResidentStudent*) jim creates for jim *a temporary alias* whose *ST* corresponds to the *cast type* (*ResidentStudent*).
  - Alias rs of *ST* **ResidentStudent** is then created via an assignment.
    Note. jim's *ST* always remains **Student**.

- *dynamic binding* : After the *cast* , **L3** will execute the correct version of setPremiumRate (∵ *DT* of rs is *ResidentStudent*).

Reference Type Casting: Motivation (1.2)

```
ST: ResidentStudent                                          ST: Student
⏞                    valid substitution                        ⏞
ResidentStudent rs        =         (ResidentStudent)    jim  ;
                                    ⏟
                                    an alias whose ST is ResidentStudent
```
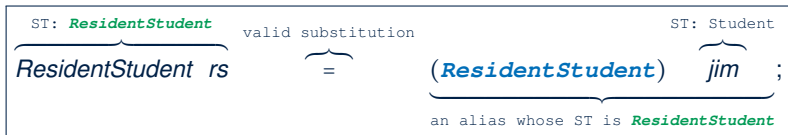
○ Variable `rs` is declared of *static type* (*ST*) ResidentStudent.
○ Variable `jim` is declared of *ST* Student.
○ The cast │(*ResidentStudent*) jim│ creates for `jim` a **temporary alias**,
  whose *ST* corresponds to the *cast type* (ResidentStudent).
    ⇒ Such a cast makes the assignment <u>valid</u>.
    ∵ RHS's *ST* (ResidentStudent) is a <u>descendant</u> of LHS's *ST*
    (ResidentStudent).
    ⇒ The assignment creates an <u>alias</u> rs with *ST* ResidentStudent.
○ **No** new object is created.
    Only an *alias* <u>rs</u> with a different *ST* (ResidentStudent) is created.
○ After the assignment, `jim`'s *ST* **remains** Student.

```
1  SmartPhone aPhone = new IPhone13Pro();
2  IPhone13Pro forHeeyeon = aPhone;
3  forHeeyeon.facetime(1.5);
```

- **L1** is *legal*: IPhone13Pro is a **descendant class** of the *static type* of aPhone (i.e., SmartPhone).
- **L2** is *illegal*: aPhone's *ST* (i.e., SmartPhone) is *__not__* a **descendant class** of forHeeyeon's *ST* (i.e., IPhone13Pro).

  Java compiler is *unable to infer* that aPhone's *dynamic type* in **L2** is IPhone13Pro!

- Force the Java compiler to believe so via a cast in **L2**:

  ```
  IPhone13Pro forHeeyeon = (IPhone13Pro) aPhone;
  ```

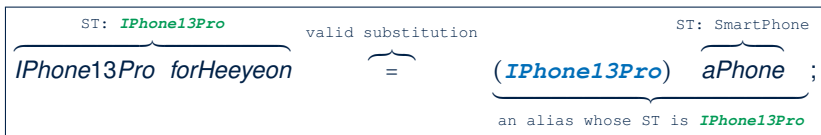 - The cast (*IPhone13Pro*) aPhone creates for aPhone *a temporary alias* whose *ST* corresponds to the *cast type* (*IPhone13Pro*).
  - Alias forHeeyeon of *ST* **IPhone13Pro** is then created via an assignment. **Note**. aPhone's *ST* always remains **SmartPhone**.

- *dynamic binding* : After the *cast* , **L3** will execute the correct version of facetime (∵ *DT* of forHeeyeon is *IPhone13Pro*).

Reference Type Casting: Motivation (2.2)

$$\overbrace{IPhone13Pro \; forHeeyeon}^{\text{ST: } \textbf{\textit{IPhone13Pro}}} \quad \overbrace{=}^{\text{valid substitution}} \quad \underbrace{(\textbf{\textit{IPhone13Pro}}) \quad \overbrace{aPhone}^{\text{ST: SmartPhone}}}_{\text{an alias whose ST is } \textbf{\textit{IPhone13Pro}}} \; ;$$

- Variable `forHeeyeon` is declared of *static type* (*ST*) IPhone13Pro.
- Variable `aPhone` is declared of *ST* SmartPhone.
- The cast `(IPhone13Pro) aPhone` creates for `aPhone` a **temporary alias**, whose *ST* corresponds to the *cast type* (IPhone13Pro).
    - ⇒ Such a cast makes the assignment <u>valid</u>.
    - ∵ RHS's *ST* (IPhone13Pro) is a <u>descendant</u> of LHS's *ST* (IPhone13Pro).
    - ⇒ The assignment creates an <u>alias</u> forHeeyeon with *ST* IPhone13Pro.
- **No** new object is created.

    Only an *alias* `forHeeyeon` with a different *ST* (IPhone13Pro) is created.
- After the assignment, `aPhone`'s *ST* **<u>remains</u>** SmartPhone.

Type Cast: Named or Anonymous

Named Cast: Use intermediate variable to store the cast result.

```
SmartPhone aPhone = new IPhone13Pro();
IOS forHeeyeon = (IPhone13Pro) aPhone;
forHeeyeon.facetime();
```

Anonymous Cast: Use the cast result directly.

```
SmartPhone aPhone = new IPhone13Pro();
((IPhone13Pro) aPhone).facetime();
```

Common Mistake:

```
1  SmartPhone aPhone = new IPhone13Pro();
2  (IPhone13Pro) aPhone.facetime();
```

L2 ≡ `(IPhone13Pro) (aPhone.facetime())` : Call, then cast.

⇒ This does **not** compile ∵ `facetime()` is **not** declared in the *static type* of aPhone (SmartPhone).

Notes on Type Cast (1)

- Given variable **v** of *static type* ST_v, it is **compilable** to cast **v** to
  C, as long as **C** is an **ancestor** or **descendant** of ST_v.
- Without cast, we can **only** call methods defined in ST_v on *v*.
- Casting *v* to **C** creates for *v* an alias with *ST* **C**.
  ⇒ All methods that are defined in **C** can be called.

```
Android myPhone = new GalaxyS21Plus();
/* can call methods declared in Android on myPhone
 * dial, surfweb, skype  ✓   sideSync  × */
SmartPhone sp = (SmartPhone) myPhone;
/* Compiles OK ∵ SmartPhone is an ancestor class of Android
 * expectations on sp narrowed to methods in SmartPhone
 * sp.dial, sp.surfweb  ✓   sp.skype, sp.sideSync  × */
GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
/* Compiles OK ∵ GalaxyS21Plus is a descendant class of Android
 * expectations on ga widened to methods in GalaxyS21Plus
 * ga.dial, ga.surfweb, ga.skype, ga.sideSync  ✓ */
```

Reference Type Casting: Danger (1)

```
1  Student jim = new NonResidentStudent("J. Davis");
2  ResidentStudent rs = (ResidentStudent) jim;
3  rs.setPremiumRate(1.5);
```

- **L1** is *legal*: NonResidentStudent is a **descendant** of the static type of jim (Student).
- **L2** is *legal* (where the cast type is ResidentStudent):
  - cast type is **descendant** of jim's ST (Student).
  - cast type is **descendant** of rs's ST (ResidentStudent).
- **L3** is *legal* ∵ setPremiumRate is in rs' **ST** ResidentStudent.
- Java compiler is *unable to infer* that jim's *dynamic type* in **L2** is actually NonResidentStudent.
- Executing **L2** will result in a *ClassCastException*.
  ∵ Attribute premiumRate (expected from a *ResidentStudent*) is *undefined* on the *NonResidentStudent* object being cast.

Reference Type Casting: Danger (2)

```
1  SmartPhone aPhone = new GalaxyS21Plus();
2  IPhone13Pro forHeeyeon = (IPhone13Pro) aPhone;
3  forHeeyeon.quickTake();
```

- **L1** is *legal*: GalaxyS21Plus is a **descendant** of the static type of aPhone (SmartPhone).
- **L2** is *legal* (where the cast type is Iphone6sPlus):
  ○ cast type is **descendant** of aPhone's ST (SmartPhone).
  ○ cast type is **descendant** of forHeeyeon's ST (IPhone13Pro).
- **L3** is *legal* ∵ quickTake is in forHeeyeon' **ST** IPhone13Pro.
- Java compiler is *unable to infer* that aPhone's *dynamic type* in **L2** is actually GalaxyS21Plus.
- Executing **L2** will result in a *ClassCastException* .
  ∵ Methods facetime, quickTake (expected from an *IPhone13Pro*) is *undefined* on the *GalaxyS21Plus* object being cast.

Notes on Type Cast (2.1)

Given a variable *v* of static type ST_v and dynamic type DT_v:

- `(C) v` is *compilable* if C is ST_v's **ancestor** or **descendant**.
- Casting v to C's **ancestor**/**descendant** **narrows**/**widens** expectations.
- However, being *compilable* does not guarantee **runtime-error-free**!

```
1  SmartPhone myPhone = new Samsung();
2  /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3  GalaxyS21Plus ga = (GalaxyS21Plus) myPhone;
4  /* Compiles OK ∵ GalaxyS21Plus is a descendant class of SmartPhone
5   * can now call methods declared in GalaxyS21Plus on ga
6   * ga.dial, ga.surfweb, ga.skype, ga.sideSync  ✓ */
```

- Type cast in **L3** is *compilable*.

- Executing **L3** will cause *ClassCastException*.

  L3: myPhone's *DT* Samsung cannot meet expectations of the
  temporary *ST* GalaxyS21Plus (e.g., sideSync).

Notes on Type Cast (2.2)

Given a variable *v* of static type ST_v and dynamic type DT_v:

- `(C) v` is *compilable* if `C` is ST_v's **ancestor** or **descendant**.
- Casting `v` to `C`'s ***ancestor***/***descendant*** ***narrows***/***widens*** expectations.
- However, being *compilable* does not guarantee ***runtime-error-free***!

```
1  SmartPhone myPhone = new Samsung();
2  /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3  IPhone13Pro ip = (IPhone13Pro) myPhone;
4  /* Compiles OK ∵ IPhone13Pro is a descendant class of SmartPhone
5   * can now call methods declared in IPhone13Pro on ip
6   * ip.dial, ip.surfweb, ip.facetime,  ip.quickTake   ✓ */
```

- Type cast in **L3** is *compilable* .

- Executing **L3** will cause *ClassCastException* .

  L3: myPhone's *DT* Samsung cannot meet expectations of the
  temporary *ST* IPhone13Pro (e.g., quickTake).

Notes on Type Cast (2.3)

A cast `(C) v` is *compilable* and ***runtime-error-free*** if `C` is located along the **ancestor path** of DT_v.

e.g., Given `**Android** myPhone = new **Samsung**();`

- Cast `myPhone` to a class along the **ancestor path** of its ***DT Samsung***.
- Casting `myPhone` to a class with more expectations than its ***DT Samsung*** (e.g., `GalaxyS21Plus`) will cause `ClassCastException`.
- Casting `myPhone` to a class irrelevant to its ***DT Samsung*** (e.g., `HuaweiMate40Pro`) will cause `ClassCastException`.

Required Reading:
Static Types, Dynamic Types, Casts

https://www.eecs.yorku.ca/~jackie/teaching/
lectures/2021/F/EECS2030/notes/EECS2030 F21
Notes Static Types Cast.pdf

```
class A { }
class B extends A { }
class C extends B { }
class D extends A { }
```

```
1  B b = new C();
2  D d = (D) b;
```

- After **L1**:
  - *ST* of b is B
  - *DT* of b is C
- Does **L2** compile?   [NO]
  - ∵ cast type D is neither an ancestor nor a descendant of b's *ST* B
- Would `D d = (D) ((A) b)` fix **L2**?   [YES]
  - ∵ cast type D is an ancestor of b's cast, temporary *ST* A
- ClassCastException when executing this fixed **L2**? [YES]
  - ∵ cast type D is not an ancestor of b's *DT* C

Reference Type Casting: Runtime Check (1)

```
1  Student jim = new NonResidentStudent("J. Davis");
2  if (jim instanceof ResidentStudent) {
3    ResidentStudent rs = ( ResidentStudent ) jim;
4    rs.setPremiumRate(1.5);
5  }
```

- **L1** is *legal*: NonResidentStudent is a **descendant class** of the **static type** of jim (i.e., Student).
- **L2** checks if jim's **DT** is a <u>descendant</u> of ResidentStudent.
    - *FALSE* ∵ jim's *dynamic type* is NonResidentStudent!
- **L3** is *legal*: jim's cast type (i.e., ResidentStudent) is a **descendant class** of rs's **ST** (i.e., ResidentStudent).
- **L3** will not be executed at runtime, hence no ClassCastException, thanks to the check in **L2**!

Reference Type Casting: Runtime Check (2)

```
1  SmartPhone aPhone = new GalaxyS21Plus();
2  if (aPhone instanceof IPhone13Pro ) {
3    IOS forHeeyeon = ( IPhone13Pro ) aPhone;
4    forHeeyeon.facetime();
5  }
```

- **L1** is *legal*: GalaxyS21Plus is a **descendant class** of the static type of aPhone (i.e., SmartPhone).
- **L2** checks if aPhone's *DT* is a <u>descendant</u> of IPhone13Pro.
  FALSE ∵ aPhone's *dynamic type* is GalaxyS21Plus!
- **L3** is *legal*: aPhone's cast type (i.e., IPhone13Pro) is a **descendant class** of forHeeyeon's *static type* (i.e., IOS).
- **L3** will not be executed at runtime, hence no ClassCastException, thanks to the check in **L2**!

Notes on the `instanceof` Operator (1)

Given a reference variable `v` and a class `C`, you write

$$v \texttt{ instanceof } C$$

to check if the ***dynamic type*** of `v`, <u>at the moment</u> of being checked, is a **descendant class** of `C` (so that $(C)\ v$ is <u>safe</u>).

```
SmartPhone myPhone = new Samsung();
println(myPhone instanceof Android);
/* true ∵ Samsung is a descendant of Android */
println(myPhone instanceof Samsung);
/* true ∵ Samsung is a descendant of Samsung */
println(myPhone instanceof GalaxyS21);
/* false ∵ Samsung is not a descendant of GalaxyS21 */
println(myPhone instanceof IOS);
/* false ∵ Samsung is not a descendant of IOS */
println(myPhone instanceof IPhone13Pro);
/* false ∵ Samsung is not a descendant of IPhone13Pro */
```

⇒ ***Samsung*** is the most specific type which `myPhone` can be **safely** cast to.

Notes on the `instanceof` Operator (2)

Given a reference variable v and a class C,

v **instanceof** C checks if the *dynamic type* of v, at the moment of being checked, is a descendant class of C.

```
1  SmartPhone myPhone = new Samsung();
2  /* ST of myPhone is SmartPhone; DT of myPhone is Samsung */
3  if(myPhone instanceof Samsung) {
4    Samsung samsung = (Samsung) myPhone;
5  }
6  if(myPhone instanceof GalaxyS21Plus) {
7    GalaxyS21Plus galaxy = (GalaxyS21Plus) myPhone;
8  }
9  if(myphone instanceof HuaweiMate40Pro) {
10   Huawei hw = (HuaweiMate40Pro) myPhone;
11 }
```

- **L3** evaluates to *true*.   [*safe* to cast]
- **L6** and **L9** evaluate to *false*.   [*unsafe* to cast]
  This prevents **L7** and **L10**, causing ClassCastException if executed, from being executed.

Static Types, Casts, Polymorphism (1.1)

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1   SmartPhone  sp = new IPhone13Pro();    ✓
2   sp.dial();    ✓
3   sp.facetime();    ×
4   sp.quickTake();    ×
```

Static type of *sp* is SmartPhone

⇒ can only call methods defined in SmartPhone on *sp*

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1  IOS  ip = new IPhone13Pro();   ✓
2  ip.dial();   ✓
3  ip.facetime();   ✓
4  ip.quickTake();   ×
```

Static type of *ip* is IOS

⇒ can only call methods defined in IOS on *ip*

```
class SmartPhone {
 void dial() { ... }
}
class IOS extends SmartPhone {
 void facetime() { ... }
}
class IPhone13Pro extends IOS {
 void quickTake() { ... }
}
```

```
1  IPhone13Pro ip6sp = new IPhone13Pro();   ✓
2  ip6sp.dial();   ✓
3  ip6sp.facetime();   ✓
4  ip6sp.quickTake();   ✓
```

Static type of *ip6sp* is IPhone13Pro

⇒ can call all methods defined in IPhone13Pro on *ip6sp*

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone13Pro extends IOS {
  void quickTake() { ... }
}
```

```
1  SmartPhone  sp = new IPhone13Pro();     ✓
2  ( (IPhone13Pro)  sp).dial();     ✓
3  ( (IPhone13Pro)  sp).facetime();   ✓
4  ( (IPhone13Pro)  sp).quickTake();   ✓
```

L4 is equivalent to the following two lines:

```
IPhone13Pro ip6sp =  (IPhone13Pro)  sp;
ip6sp.quickTake();
```

Static Types, Casts, Polymorphism (2)

Given a reference variable declaration

```
C v;
```

- *Static type* of reference variable *v* is class *C*
- A method call `v.m` is valid if *m* is a method **defined** in class *C*.
- Despite the *dynamic type* of *v*, you are only allowed to call methods that are defined in the *static type* C on *v*.
- If you are certain that *v*'s *dynamic type* can be expected **more** than its *static type*, then you may use an insanceof check and a cast.

```
Course eecs2030 = new Course("EECS2030", 500.0);
Student s = new ResidentStudent("Jim");
s.register(eecs2030);
if(s instanceof ResidentStudent) {
  ( (ResidentStudent) s).setPremiumRate(1.75);
  System.out.println(( (ResidentStudent) s).getTuition());
}
```

Polymorphism: Method Parameters (1)

```
1  class StudentManagementSystem {
2    Student [] ss; /* ss[i] has static type Student */ int c;
3    void addRS(ResidentStudent rs) { ss[c] = rs; c ++; }
4    void addNRS(NonResidentStudent nrs) { ss[c] = nrs; c++; }
5    void addStudent(Student s) { ss[c] = s; c++; } }
```

- **L3**: ss[c] = rs is valid. ∵ RHS's ST ResidentStudent is a *descendant class* of LHS's ST Student.
- Say we have a StudentManagementSystem object sms:
  - ◦ $\boxed{\texttt{sms.addRS(o)}}$ attempts the following assignment (recall call by value), which replaces parameter rs by a copy of argument o:

    ```
    rs = o;
    ```

  - ◦ Whether this argument passing is valid depends on o's *static type*.
- In the signature of a method m, if the type of a parameter is class C, then we may call method m by passing objects whose *static types* are C's *descendants*.

In the `StudentManagementSystemTester`:

```
Student s1 = new Student();
Student s2 = new ResidentStudent();
Student s3 = new NonResidentStudent();
ResidentStudent rs = new ResidentStudent();
NonResidentStudent nrs = new NonResidentStudent();
StudentManagementSystem sms = new StudentManagementSystem();
sms.addRS(s1);    ×
sms.addRS(s2);    ×
sms.addRS(s3);    ×
sms.addRS(rs);    ✓
sms.addRS(nrs);   ×
sms.addStudent(s1);    ✓
sms.addStudent(s2);    ✓
sms.addStudent(s3);    ✓
sms.addStudent(rs);    ✓
sms.addStudent(nrs);   ✓
```

Polymorphism: Method Parameters (2.2)

In the `StudentManagementSystemTester`:

```
1  Student s = new Student("Stella");
2  /* s' ST: Student; s' DT: Student */
3  StudentManagementSystem sms = new StudentManagementSystem();
4  sms.addRS(s); ×
```

- **L4** *compiles* with a cast: `sms.addRS((ResidentStudent) s)`
  - *Valid* cast ∵ (ResidentStudent) is a <u>descendant</u> of s' **ST**.
  - *Valid* call ∵ s' temporary **ST** (ResidentStudent) is now a <u>descendant</u> class of addRS's parameter rs' **ST** (ResidentStudent).
- But, there will be a `ClassCastException` at runtime!
  ∵ s' **DT** (Student) is **not** a <u>descendant</u> of ResidentStudent.
- We should have written:

```
if(s instanceof ResidentStudent) {
  sms.addRS((ResidentStudent) s);
}
```

The **instanceof** expression will evaluate to *false*, meaning it is *unsafe* to cast, thus preventing ClassCastException.

Polymorphism: Method Parameters (2.3)

In the `StudentManagementSystemTester`:

```
1  Student s = new NonResidentStudent("Nancy");
2  /* s' ST: Student; s' DT: NonResidentStudent */
3  StudentManagementSystem sms = new StudentManagementSystem();
4  sms.addRS(s); ×
```

- **L4** *compiles* with a cast: `sms.addRS((ResidentStudent) s)`
  - *Valid* cast ∵ (ResidentStudent) is a <u>descendant</u> of s' **ST**.
  - *Valid* call ∵ s' temporary **ST** (ResidentStudent) is now a <u>descendant</u> class of addRS's parameter rs' **ST** (ResidentStudent).
- But, there will be a `ClassCastException` at runtime!
  ∵ s' **DT** (NonResidentStudent) **not** <u>descendant</u> of ResidentStudent.
- We should have written:

```
if(s instanceof ResidentStudent) {
  sms.addRS((ResidentStudent) s);
}
```

The **instanceof** expression will evaluate to *false*, meaning it is *unsafe* to cast, thus preventing ClassCastException.

Polymorphism: Method Parameters (2.4)

In the `StudentManagementSystemTester`:

```
1  Student s = new ResidentStudent("Rachael");
2  /* s' ST: Student; s' DT: ResidentStudent */
3  StudentManagementSystem sms = new StudentManagementSystem();
4  sms.addRS(s); ×
```

- **L4** *compiles* with a cast: `sms.addRS((ResidentStudent) s)`
  - *Valid* cast ∵ (ResidentStudent) is a <u>descendant</u> of s' **ST**.
  - *Valid* call ∵ s' temporary **ST** (ResidentStudent) is now a <u>descendant</u> class of addRS's parameter rs' **ST** (ResidentStudent).
- And, there will be **no** `ClassCastException` at runtime!
  ∵ s' **DT** (ResidentStudent) is <u>descendant</u> of ResidentStudent.
- We should have written:

  ```
  if(s instanceof ResidentStudent) {
    sms.addRS((ResidentStudent) s);
  }
  ```

  The **instanceof** expression will evaluate to *true*, meaning it is *safe* to cast.

Polymorphism: Method Parameters (2.5)

In the `StudentManagementSystemTester`:

```
1  NonResidentStudent nrs = new NonResidentStudent();
2  /* ST: NonResidentStudent; DT: NonResidentStudent */
3  StudentManagementSystem sms = new StudentManagementSystem();
4  sms.addRS(nrs); ×
```

Will **L4** with a cast compile?

```
sms.addRS( (ResidentStudent)  nrs)
```

NO ∵ (ResidentStudent) is *not* a <u>descendant</u> of nrs's *ST* (NonResidentStudent).

Why Inheritance:
A Polymorphic Collection of Students

How do you define a class `StudentManagementSystem` that contains a list of *resident* and *non-resident* students?

```
class StudentManagementSystem {
  Student[] students;
  int numOfStudents;

  void addStudent(Student s) {
    students[numOfStudents] = s;
    numOfStudents ++;
  }

  void registerAll (Course c) {
    for(int i = 0; i < numberOfStudents; i ++) {
      students[i].register(c)
    }
  }
}
```

a collection of students without inheritance

Polymorphism and Dynamic Binding:
A Polymorphic Collection of Students (1)

```
1   ResidentStudent rs = new ResidentStudent("Rachael");
2   rs.setPremiumRate(1.5);
3   NonResidentStudent nrs = new NonResidentStudent("Nancy");
4   nrs.setDiscountRate(0.5);
5   StudentManagementSystem sms = new StudentManagementSystem();
6   sms.addStudent( rs ); /* polymorphism */
7   sms.addStudent( nrs ); /* polymorphism */
8   Course eecs2030 = new Course("EECS2030", 500.0);
9   sms.registerAll(eecs2030);
10  for(int i = 0; i < sms.numberOfStudents; i ++) {
11    /* Dynamic Binding:
12     * Right version of getTuition will be called */
13    System.out.println(sms.students[i]. getTuition() );
14  }
```

Polymorphism and Dynamic Binding:
A Polymorphic Collection of Students (2)

At runtime, attribute `sms.ss` is a *polymorphic* array:
- *Static type* of each item is as declared: *Student*
- *Dynamic type* of each item is a **descendant** of *Student*: *ResidentStudent*, *NonResidentStudent*
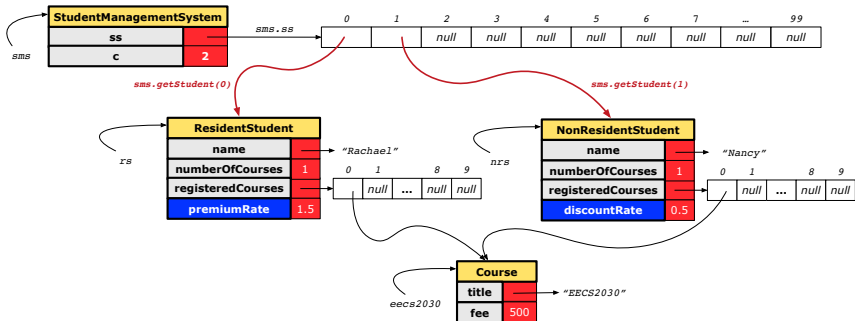
Polymorphism: Return Types (1)

```
1   class StudentManagementSystem {
2     Student[] ss; int c;
3     void addStudent(Student s) { ss[c] = s; c++; }
4     Student  getStudent(int i) {
5       Student s = null;
6       if(i < 0 || i >= c) {
7         throw new InvalidStudentIndexException("Invalid index.");
8       }
9       else {
10        s = ss[i];
11      }
12      return s;
13    } }
```

L4: Student is *static type* of getStudent's return value.

L10: ss[i]'s ST (Student) is **descendant** of s' ST (Student).

Question: What can be the *dynamic type* of s after **L10**?

Answer: All descendant classes of Student.

Polymorphism: Return Types (2)

```
1  Course eecs2030 = new Course("EECS2030", 500);
2  ResidentStudent rs = new ResidentStudent("Rachael");
3  rs.setPremiumRate(1.5); rs.register(eecs2030);
4  NonResidentStudent nrs = new NonResidentStudent("Nancy");
5  nrs.setDiscountRate(0.5); nrs.register(eecs2030);
6  StudentManagementSystem sms = new StudentManagementSystem();
7  sms.addStudent(rs); sms.addStudent(nrs);
8  Student  s  =    sms.getStudent(0)  ;  /* dynamic type of s? */

              static return type: Student
9  print(s instanceof Student && s instanceof ResidentStudent);/*true*/
10 print(s instanceof NonResidentStudent);  /* false */
11 print( s.getTuition() );/*Version in ResidentStudent called:750*/
12 ResidentStudent rs2 = sms.getStudent(0);   ×
13 s  =    sms.getStudent(1)  ;  /* dynamic type of s? */

         static return type: Student
14 print(s instanceof Student && s instanceof NonResidentStudent);/*true*/
15 print(s instanceof ResidentStudent);  /* false */
16 print( s.getTuition() );/*Version in NonResidentStudent called:250*/
17 NonResidentStudent nrs2 = sms.getStudent(1); ×
```

Polymorphism: Return Types (3)

At runtime, attribute `sms.ss` is a *polymorphic* array:

- *Static type* of each item is as declared: *Student*
- *Dynamic type* of each item is a **descendant** of *Student*: *ResidentStudent*, *NonResidentStudent*
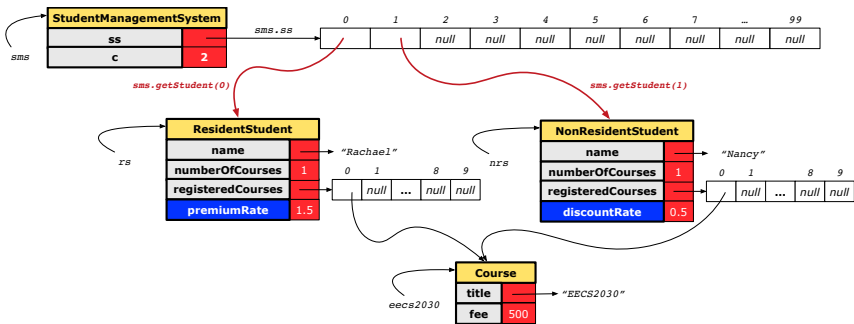
Static Type vs. Dynamic Type:
When to consider which?

- *Whether or not Java code compiles* depends only on the **static types** of relevant variables.

  ∵ Inferring the **dynamic type** statically is an *undecidable* problem that is inherently impossible to solve.

- *The behaviour of Java code being executed at runtime* (e.g., which version of method is called due to dynamic binding, whether or not a `ClassCastException` will occur, *etc.*) depends on the **dynamic types** of relevant variables.

  ⇒ Best practice is to visualize how objects are created (by drawing boxes) and variables are re-assigned (by drawing arrows).

Summary: Type Checking Rules

Code	Condition to be Type Correct
`x = y`	Is `y`'s *ST* a **descendant** of `x`'s *ST*?
`x.m(y)`	Is method `m` defined in `x`'s *ST*?
	Is `y`'s *ST* a **descendant** of `m`'s parameter's *ST*?
`z = x.m(y)`	Is method `m` defined in `x`'s *ST*?
	Is `y`'s *ST* a **descendant** of `m`'s parameter's *ST*?
	Is *ST* of `m`'s return value a **descendant** of `z`'s *ST*?
`(C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s *ST*?
`x = (C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s *ST*?
	Is `C` a **descendant** of `x`'s *ST*?
`x.m((C) y)`	Is `C` an **ancestor** or a **descendant** of `y`'s *ST*?
	Is method `m` defined in `x`'s *ST*?
	Is `C` a **descendant** of `m`'s parameter's *ST*?

Even if `(C) y` compiles OK, there will be a runtime
`ClassCastException` if `C` is not an **ancestor** of `y`'s *DT*!

Root of the Java Class Hierarchy

- Implicitly:
  - Every class is a *child/sub* class of the `Object` class.
  - The `Object` class is the *parent/super* class of every class.
- There are two useful *accessor methods* that every class *inherits* from the `Object` class:
  - `boolean equals(Object other)`
    Indicates whether some other object is "equal to" this one.
    - The default definition inherited from `Object`:

    ```
    boolean equals(Object other) {
      return (this == other); }
    ```

  - `String toString()`
    Returns a string representation of the object.
- Very often when you define new classes, you want to *redefine* / *override* the inherited definitions of `equals` and `toString`.

Overriding and Dynamic Binding (1)

`Object` is the common parent/super class of every class.

- Every class inherits the **default version** of `equals`
- Say a reference variable *v* has **dynamic type** *D*:
  - **Case 1** *D overrides* `equals`
    ⇒ `v.equals(...)` invokes the **overridden version** in *D*
  - **Case 2** *D* does *not override* `equals`
    Case 2.1 At least one ancestor classes of *D override* `equals`
    ⇒ `v.equals(...)` invokes the **overridden version** in the **closest ancestor class**
    Case 2.2 No ancestor classes of *D override* `equals`
    ⇒ `v.equals(...)` invokes **default version** inherited from `Object`.
- Same principle applies to the `toString` method, and all overridden methods in general.

Overriding and Dynamic Binding (2.1)
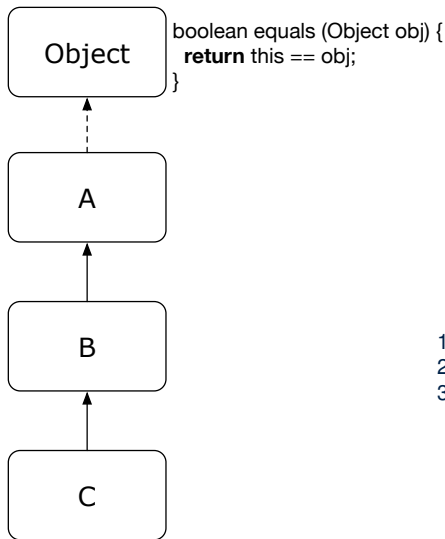
Object

```
boolean equals (Object obj) {
  return this == obj;
}
```

A

B

C

```
class A {
  /*equals not overridden*/
}
class B extends A {
  /*equals not overridden*/
}
class C extends B {
  /*equals not overridden*/
}
```

```
1  Object c1 = new C();
2  Object c2 = new C();
3  println(c1.equals(c2));
```

L3 calls which version of equals?   [Object]

Overriding and Dynamic Binding (2.2)

```
Object    boolean equals (Object obj) {
            return this == obj;
          }
```
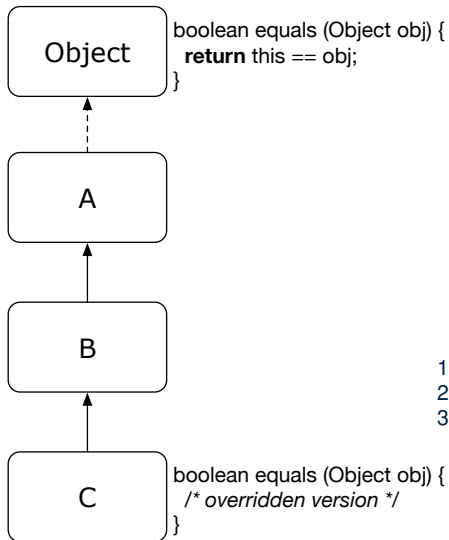
```
class A {
  /*equals not overridden*/
}
class B extends A {
  /*equals not overridden*/
}
class C extends B {
  boolean equals(Object obj) {
    /* overridden version */
  }
}
```

```
A
```

```
B
```

```
1  Object c1 = new C();
2  Object c2 = new C();
3  println(c1.equals(c2));
```

```
C    boolean equals (Object obj) {
       /* overridden version */
     }
```

L3 calls which version of equals?   [C]

```
class A {
  /*equals not overridden*/
}
class B extends A {
  boolean equals(Object obj) {
    /* overridden version */
  }
}
class C extends B {
  /*equals not overridden*/
}
```

Object — boolean equals (Object obj) {
  return this == obj;
}

B — boolean equals (Object obj) {
  /* overridden version */
}

```
1  Object c1 = new C();
2  Object c2 = new C();
3  println(c1.equals(c2));
```

L3 calls which version of equals? [B]

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
Point@677327b6
```

- Implicitly, the `toString` method is called inside the `println` method.
- By default, the address stored in `p1` gets printed.
- We need to *redefine* / *override* the `toString` method, inherited from the `Object` class, in the `Point` class.

```
class Point {
  double x;
  double y;
  public String toString() {
    return "(" + this.x + ", " + this.y + ")";
  }
}
```

After redefining/overriding the `toString` method:

```
Point p1 = new Point(2, 4);
System.out.println(p1);
```

```
(2, 4)
```

Exercise: Override the `equals` and `toString` methods for the `ResidentStudent` and `NonResidentStudent` classes.

Beyond this lecture...

- Implement the *inheritance hierarchy* of **Students** and reproduce all lecture examples.

- Implement the *inheritance hierarchy* of **Smart Phones** and reproduce all lecture examples.

  Hints. Pay attention to:

  ∘ *Valid*? *Compiles*?
  ∘ *ClassCastException*?

- Study the `ExampleTypeCasts` example: draw the *inheritance hierarchy* and experiment with the various <u>substitutions</u> and casts.

Index (4)

Index (7)

Index (8)

Index (9)

Abstract Classes and Interfaces

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

Chen-Wei Wang

Learning Outcomes

This module is designed to help you learn about:

- What an *abstract* method and an *abstract* class are
- What an *interface* is
- Reinforce: *Polymorphism* and *dynamic binding*
- *When* to use abstract classes vs. interfaces?

Abstract Class (1)

Problem: A polygon may be either a triangle or a rectangle. Given a polygon, we may either
- ***Grow*** its shape by incrementing the size of each of its sides;
- Compute and return its ***perimeter***; or
- Compute and return its ***area***.

- For a rectangle with *length* and *width*, its area is *length* × *width*.
- For a triangle with sides *a*, *b*, and *c*, its area, according to Heron's formula, is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a+b+c}{2}$$

- How would you solve this problem in Java, while *minimizing code duplicates* ?

Abstract Class (2)

```java
public abstract class Polygon {
 double[] sides;
 Polygon(double[] sides) { this.sides = sides; }
 void grow() {
   for(int i = 0; i < sides.length; i ++) { sides[i] ++; }
 }
 double getPerimeter() {
   double perimeter = 0;
   for(int i = 0; i < sides.length; i ++) {
    perimeter += sides[i];
   }
   return perimeter;
 }
 abstract double getArea();
}
```

- Method `getArea` not implemented and shown **header** only.
- ∴ `Polygon` cannot be used as a **dynamic type**
- Writing *new* `Polygon(...)` is forbidden!

```
public class Rectangle extends Polygon {
  Rectangle(double length, double width) {
    super(new double[4]);
    sides[0] = length; sides[1] = width;
    sides[2] = length; sides[3] = width;
  }
  double getArea() { return sides[0] * sides[1]; }
}
```

- Method `getPerimeter` is inherited from the super-class `Polygon`.
- Method `getArea` is implemented in the sub-class `Rectangle`.
- ∴ `Rectangle` can be used as a *dynamic type*
- Writing `Polygon p = new Rectangle(3, 4)` allowed!

Abstract Class (4)

```java
public class Triangle extends Polygon {
  Triangle(double side1, double side2, double side3) {
    super(new double[3]);
    sides[0] = side1; sides[1] = side2; sides[2] = side3;
  }
  double getArea() {
    /* Heron's formula */
    double s = getPerimeter() * 0.5;
    double area = Math.sqrt(
      s * (s - sides[0]) * (s - sides[1]) * (s - sides[2]));
    return area;
  }
}
```

- Method `getPerimeter` is inherited from `Polygon`.
- Method `getArea` is implemented in the sub-class `Triangle`.
- ∴ `Triangle` can be used as a *dynamic type*
- Writing `Polygon p =` *new* `Triangle(3, 4, 5)` allowed!

Abstract Class (5)

```
1   public class PolygonCollector {
2     Polygon[] polygons;
3     int numberOfPolygons;
4     PolygonCollector() { polygons = new Polygon[10]; }
5     void addPolygon(Polygon p) {
6       polygons[numberOfPolygons] = p; numberOfPolygons ++;
7     }
8     void growAll() {
9       for(int i = 0; i < numberOfPolygons; i ++) {
10        polygons[i].grow();
11      }
12    }
13  }
```

- **Polymorphism**: **Line 5** may accept as argument any object whose *static type* is `Polygon` or any of its sub-classes.
- **Dynamic Binding**: **Line 10** calls the version of `grow` inherited to the *dynamic type* of `polygons[i]`.

Abstract Class (6)

```
1   public class PolygonConstructor {
2     Polygon getPolygon(double[] sides) {
3       Polygon p = null;
4       if(sides.length == 3) {
5         p = new Triangle(sides[0], sides[1], sides[2]);
6       }
7       else if(sides.length == 4) {
8         p = new Rectangle(sides[0], sides[1]);
9       }
10      return p;
11    }
12    void grow(Polygon p) { p.grow(); }
13  }
```

- **Polymorphism**:
  - **Line 2** may accept as return value any object whose *static type* is Polygon or any of its sub-classes.
  - **Line 5** returns an object whose *dynamic type* is Triangle; **Line 8** returns an object whose *dynamic type* is Rectangle.

Abstract Class (7.1)

```
1  public class PolygonTester {
2    public static void main(String[] args) {
3      Polygon p;
4      p = new Rectangle(3, 4); /* polymorphism */
5      System.out.println(p.getPerimeter()); /* 14.0 */
6      System.out.println(p.getArea()); /* 12.0 */
7      p = new Triangle(3, 4, 5); /* polymorphism */
8      System.out.println(p.getPerimeter()); /* 12.0 */
9      System.out.println(p.getArea()); /* 6.0 */
10
11     PolygonCollector col = new PolygonCollector();
12     col.addPolygon(new Rectangle(3, 4)); /* polymorphism */
13     col.addPolygon(new Triangle(3, 4, 5)); /* polymorphism */
14     System.out.println(col.polygons[0].getPerimeter()); /* 14.0 */
15     System.out.println(col.polygons[1].getPerimeter()); /* 12.0 */
16     col.growAll();
17     System.out.println(col.polygons[0].getPerimeter()); /* 18.0 */
18     System.out.println(col.polygons[1].getPerimeter()); /* 15.0 */
```

Abstract Class (7.2)

```
1    PolygonConstructor con = new PolygonConstructor();
2    double[] recSides = {3, 4, 3, 4};  p = con.getPolygon(recSides);
3    System.out.println(p instanceof Polygon);    ✓
4    System.out.println(p instanceof Rectangle);   ✓
5    System.out.println(p instanceof Triangle);    ×
6    System.out.println(p.getPerimeter()); /* 14.0 */
7    System.out.println(p.getArea()); /* 12.0 */
8    con.grow( p );
9    System.out.println(p.getPerimeter()); /* 18.0 */
10   System.out.println(p.getArea()); /* 20.0 */
11   double[] triSides = {3, 4, 5};  p = con.getPolygon(triSides);
12   System.out.println(p instanceof Polygon);    ✓
13   System.out.println(p instanceof Rectangle);   ×
14   System.out.println(p instanceof Triangle);    ✓
15   System.out.println(p.getPerimeter()); /* 12.0 */
16   System.out.println(p.getArea()); /* 6.0 */
17   con.grow( p );
18   System.out.println(p.getPerimeter()); /* 15.0 */
19   System.out.println(p.getArea()); /* 9.921 */
20   } }
```

Abstract Class (8)

- An *abstract class* :
  - Typically has **at least one** method with no implementation body
  - May define common implementations inherited to **sub-classes**.
- Recommended to use an *abstract class* as the *static type* of:
  - A *variable*
    e.g., `Polygon p`
  - A *method parameter*
    e.g., `void grow(Polygon p)`
  - A *method return value*
    e.g., `Polygon getPolygon(double[] sides)`
- It is forbidden to use an *abstract class* as a *dynamic type*
  e.g., `Polygon p = new Polygon(...)` is not allowed!
- Instead, create objects whose *dynamic types* are descendant classes of the *abstract class* ⇒ Exploit *dynamic binding* !
  e.g., `Polygon p = con.getPolygon(recSides)`
  This is is as if we did `Polygon p = new Rectangle(...)`

Interface (1.1)

- We may implement `Point` using two representation systems:
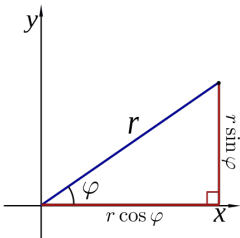
- The *Cartesian system* stores the *absolute* positions of `x` and `y`.
- The *Polar system* stores the *relative* position: the angle (in radian) `phi` and distance `r` from the origin (0.0).
- As far as users of a `Point` object `p` is concerned, being able to call `p.getX()` and `p.getY()` is what matters.
- How `p.getX()` and `p.getY()` are internally computed, depending on the *dynamic type* of `p`, do not matter to users.

Interface (1.2)

Recall: $sin30° = \frac{1}{2}$ and $cos30° = \frac{1}{2} \cdot \sqrt{3}$
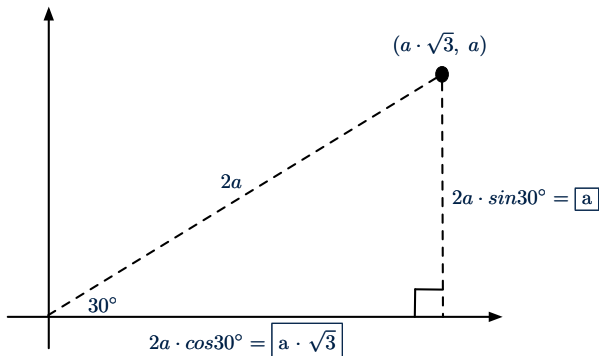
We consider the same point represented differently as:

- $r = 2a$, $\psi = 30°$ [polar system]
- $x = 2a \cdot cos30° = a \cdot \sqrt{3}$, $y = 2a \cdot sin30° = a$ [cartesian system]

Interface (2)

```
public interface Point {
  public double getX();
  public double getY();
}
```

- An interface Point defines how users may access a point: either get its *x* coordinate or its *y* coordinate.
- Methods getX and getY similar to getArea in Polygon, have no implementations, but *headers* only.
- ∴ Point cannot be used as a *dynamic type*
- Writing *new* Point(...) is forbidden!

Interface (3)

```
public class CartesianPoint implements Point {
  private double x;
  private double y;
  public CartesianPoint(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() { return x; }
  public double getY() { return y; }
}
```

- CartesianPoint is a possible implementation of Point.
- Attributes x and y declared according to the *Cartesian system*
- All method from the interface Point are implemented in the sub-class CartesianPoint.
- ∴ CartesianPoint can be used as a *dynamic type*
- Point p = *new* CartesianPoint(3, 4) allowed!

Interface (4)

```
public class PolarPoint implements Point {
  private double phi;
  private double r;
  public PolarPoint(double r, double phi) {
    this.r = r;
    this.phi = phi;
  }
  public double getX() { return Math.cos(phi) * r; }
  public double getY() { return Math.sin(phi) * r; }
}
```

- PolarPoint is a possible implementation of Point.
- Attributes phi and r declared according to the *Polar system*
- All method from the interface Point are implemented in the sub-class PolarPoint.
- ∴ PolarPoint can be used as a *dynamic type*
- Point p = *new* PolarPoint(3, $\frac{\pi}{6}$) allowed! [360° = 2π]

```
1   public class PointTester {
2    public static void main(String[] args) {
3      double A = 5;
4      double X = A * Math.sqrt(3);
5      double Y = A;
6      Point p;
7      p = new CartisianPoint(X, Y); /* polymorphism */
8      print("(" + p.getX() + ", " + p.getY() + ")"); /* dyn. bin. */
9      p = new PolarPoint(2 * A, Math.toRadians(30)); /* polymorphism */
10     print("(" + p.getX() + ", " + p.getY() + ")"); /* dyn. bin. */
11    }
12  }
```

- **Lines 7 and 9** illustrate *polymorphism*, how?
- **Lines 8 and 10** illustrate *dynamic binding*, how?

Interface (6)

- An *interface* :
  - Has **all** its methods with no implementation bodies.
  - Leaves complete freedom to its *implementors*.
- Recommended to use an *interface* as the *static type* of:
  - A *variable*
    e.g., Point p
  - A *method parameter*
    e.g., void moveUp(Point p)
  - A *method return value*
    e.g., Point getPoint(double v1, double v2, boolean isCartesian)
- It is forbidden to use an *interface* as a *dynamic type*
  e.g., Point p = new Point(...) is not allowed!
- Instead, create objects whose *dynamic types* are descendant classes of the *interface* ⇒ Exploit *dynamic binding* !

Abstract Classes vs. Interfaces: When to Use Which?

- Use *interfaces* when:
  - There is a *common set of functionalities* that can be implemented via *a variety of strategies*.
    e.g., Interface `Point` declares headers of `getX()` and `getY()`.
  - Each descendant class represents a different implementation strategy for the same set of functionalities.
  - `CartesianPoint` and `PolarPoinnt` represent different strategies for supporting `getX()` and `getY()`.
- Use *abstract classes* when:
  - *Some (not all) implementations can be shared* by descendants, and *some (not all) implementations cannot be shared*.
    e.g., Abstract class `Polygon`:
    - Defines implementation of `getPerimeter`, to be shared by `Rectangle` and `Triangle`.
    - Declares header of `getArea`, to be implemented by `Rectangle` and `Triangle`.

Beyond this lecture. . .

Study the `ExampleAbstractClasses` and
`ExampleInterfaces` source code:

○ Draw the ***inheritance hierarchy*** based on the class declarations
○ Use the ***debugger*** to step into the various method calls (e.g.,
`getArea()` of Polygon, `getX()` of Point) to see which <u>version</u> of
the method gets executed (i.e., ***dynamic binding***).

Index (1)

Index (2)

Generics in Java

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

CHEN-WEI WANG

This module is designed to help you learn about:

1. A *general* collection `Object[]`: storage vs. retrieval

2. A *generic* collection `E[]`: storage vs. retrieval

3. Reinforce: Polymorphism, *Type Casting*, `instanceof` checks

```java
public class Book {
  private String[] names;
  private Object[] records;
  /* add a name-record pair to the book */
  public void add (String name, Object record) { ... }
  /* return the record associated with a given name */
  public Object get (String name) { ... } }
```

Question: Which line has a type error?

```java
Date birthday; String phoneNumber;
Book b; boolean isWednesday;
b = new Book();
phoneNumber = "416-67-1010";
b.add ("Suyeon", phoneNumber);
birthday = new Date(1975, 4, 10);
b.add ("Yuna", birthday);
isWednesday = b.get("Yuna").getDay() == 4;
```

Motivating Example: Observations (1)

- In the `Book` class:
  - By declaring the attribute

    ```
    Object[] records
    ```

    We meant that each book instance may store any object whose *static type* is a <mark>*descendant class*</mark> of `Object`.
  - Accordingly, from the return type of the `get` method, we only know that the returned record is an `Object`, but not certain about its *dynamic type* (e.g., `Date`, `String`, *etc.*).
    ∴ a record retrieved from the book, e.g., `b.get("Yuna")`, may only be called upon methods in its *static type* (i.e,. `Object`).
- In the tester code of the `Book` class:
  - In **Line 1**, the *static types* of variables `birthday` (i.e., `Date`) and `phoneNumber` (i.e., `String`) are <mark>*descendant classes*</mark> of `Object`.
  - So, **Line 5** and **Line 7** compile.

In a *polymorphic collection*, *dynamic types* of stored objects (e.g., phoneNumber and birthday) need not be the same.

○ Methods <u>expected</u> on the *dynamic types* (e.g., method getDay of class Date) may be new methods **not** inherited from Object.

○ This is why **Line 8** would fail to compile, and may be fixed using an explicit *cast* :

```
isWednesday = ((Date) b.get("Yuna")).getDay() == 4;
```

○ But what if the *dynamic type* of the returned object is not a Date?

```
isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
```

○ To avoid such a ClassCastException at runtime, we need to check its *dynamic type* before performing a cast:

```
if (b.get("Suyeon") instanceof Date) {
  isWednesday = ((Date) b.get("Suyeon")).getDay() == 4;
}
```

Motivating Example: Observations (2.1)

- It seems: Combining **`instanceof`** checks & **type casts** works.
- Can you see any potential problem(s) w.r.t. the **Single-Choice** design principle?
- **Hints:** What happens when you have a large number of records of distinct **dynamic types** stored in the book (e.g., `Date`, `String`, `Person`, `Account`, *etc.*)?

Imagine that the tester code (or an application) stores 100 different record objects into the book.

○ All of these records are of *static type* `Object`, but of distinct *dynamic types*.

```
Object rec1 = new C1(); b.add(..., rec1);
Object rec2 = new C2(); b.add(..., rec2);
...
Object rec100 = new C100(); b.add(..., rec100);
```

where classes ***C1*** to ***C100*** are **descendant classes** of `Object`.

○ **Every time** you retrieve a record from the book, you need to check "exhaustively" on its *dynamic type* before calling some method(s).

```
Object rec = b.get("Jim");
if (rec instanceof C1) { ((C1) rec).m1; }
...
else if (rec instanceof C100) { ((C100) rec).m100; }
```

○ Writing out this list multiple times is tedious and error-prone!

Motivating Example: Observations (3)

We need a solution that:
- Saves us from explicit `instanceof` checks and type casts
- Eliminates the occurrences of `ClassCastException`

As a sketch, this is how the solution looks like:
- When the user declares a `Book` object `b`, they must
  commit to the kind of record that `b` stores at runtime .
  e.g., `b` stores either `Date` objects only or `String` objects only,
  but ***not a mix***.
- When attempting to **store** a new record object `rec` into `b`, what
  if `rec`'s *static type* is not a **descendant class** of the type of
  book that the user previously commits to?
  ⇒ A *compilation error*
- When attempting to **retrieve** a record object from `b`, there is
  no longer a need to check and cast .
  ∵ ***Static types*** of all records in `b` are guaranteed to be the same.

Parameters

- In mathematics:
  - The same *function* is applied with different *argument values*.
    e.g., `2 + 3`, `1 + 1`, `10 + 101`, *etc.*
  - We *generalize* these instance applications into a definition.
    e.g., $+ : (\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}$ is a function that takes two integer *parameters* and returns an integer.
- In Java programming:
  - We want to call a *method*, with different *argument values*, to achieve a similar goal.
    e.g., `acc.deposit(100)`, `acc.deposit(23)`, *etc.*
  - We generalize these possible method calls into a definition.
    e.g., In class `Account`, a method `void deposit(int amount)` takes one integer *parameter*.
- When you design a mathematical function or a Java method, always consider the list of *parameters*, each of which representing a set of possible *argument values*.

```java
class Book <E> {
  private String[] names;
  private E[] records;
  /* add a name-record pair to the book */
  public void add (String name, E record) { ... }
  /* return the record associated with a given name */
  public E get (String name) { ... } }
```

Question: Which line has a type error?

```
1  Date birthday; String phoneNumber;
2  Book<Date> b; boolean isWednesday;
3  b = new Book<Date>();
4  phoneNumber = "416-67-1010";
5  b.add ("Suyeon", phoneNumber);
6  birthday = new Date(1975, 4, 10);
7  b.add ("Yuna", birthday);
8  isWednesday = b.get("Yuna").getDay() == 4;
```

Java Generics: Observations

- In class `Book`:
  - At the class level, we *parameterize the type of records* that an instance of book may store: `class Book< E >`

    where E is the name of a type parameter, which should be *instantiated* when the user declares an instance of `Book`.
  - Every occurrence of `Object` (the most general type of records) is replaced by E.
  - As soon as E at the class level is committed to some known type (e.g., `Date`, `String`, *etc.*), every occurrence of E will be replaced by that type.
- In the tester code of `Book`:
  - In **Line 2**, we commit that the book `b` will store `Date` objects only.
  - **Line 5** now fails to compile.   [`String` is <u>not</u> `Date`'s descendant]
  - **Line 7** still compiles.
  - **Line 8** does *not need* any instance check and type cast, and does *not cause* any `ClassCastException`.
    - ∵ Only `Date` objects were allowed to be stored.

Example Generic Classes: `ArrayList`

An ArrayList acts like a "resizable" array (indices start at 0).
Extra tutorial here

int	`size()`	Returns the number of elements in this list.
boolean	`add(E e)`	Appends the specified element to the end of this list.
void	`add(int index, E element)`	Inserts the specified element at the specified position in this list.
boolean	`contains(Object o)`	Returns `true` if this list contains the specified element.
E	`remove(int index)`	Removes the element at the specified position in this list.
boolean	`remove(Object o)`	Removes the first occurrence of the specified element from this list, if it is present.
int	`indexOf(Object o)`	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
E	`get(int index)`	Returns the element at the specified position in this list.

```
1    import java.util.ArrayList;
2    public class ArrayListTester {
3      public static void main(String[] args) {
4        ArrayList<String> list = new ArrayList<String>();
5        println(list.size());
6        println(list.contains("A"));
7        println(list.indexOf("A"));
8        list.add("A");
9        list.add("B");
10       println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
11       println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
12       list.add(1, "C");
13       println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
14       println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
15       list.remove("C");
16       println(list.contains("A")); println(list.contains("B")); println(list.contains("C"));
17       println(list.indexOf("A")); println(list.indexOf("B")); println(list.indexOf("C"));
18
19       for(int i = 0; i < list.size(); i ++) {
20         println(list.get(i));
21       }
22     }
23   }
```

Example Generic Classes: `HashTable`

A HashTable acts like a two-column table of (searchable) keys and values. *Extra tutorial here.*

int	**size**() Returns the number of keys in this hashtable.
boolean	**containsKey**(**Object** key) Tests if the specified object is a key in this hashtable.
boolean	**containsValue**(**Object** value) Returns true if this hashtable maps one or more keys to this value.
V	**get**(**Object** key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	**put**(**K** key, **V** value) Maps the specified key to the specified value in this hashtable.
V	**remove**(**Object** key) Removes the key (and its corresponding value) from this hashtable.

Using Generic Classes: `HashTable`

```java
import java.util.Hashtable;
public class HashTableTester {
  public static void main(String[] args) {
    Hashtable<String, String> grades = new Hashtable<String, String>();
    System.out.println("Size of table: " + grades.size());
    System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
    System.out.println("Value B+ exists: " + grades.containsValue("B+"));
    grades.put("Alan", "A");
    grades.put("Mark", "B+");
    grades.put("Tom", "C");
    System.out.println("Size of table: " + grades.size());
    System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
    System.out.println("Key Mark exists: " + grades.containsKey("Mark"));
    System.out.println("Key Tom exists: " + grades.containsKey("Tom"));
    System.out.println("Key Simon exists: " + grades.containsKey("Simon"));
    System.out.println("Value A exists: " + grades.containsValue("A"));
    System.out.println("Value B+ exists: " + grades.containsValue("B+"));
    System.out.println("Value C exists: " + grades.containsValue("C"));
    System.out.println("Value A+ exists: " + grades.containsValue("A+"));
    System.out.println("Value of existing key Alan: " + grades.get("Alan"));
    System.out.println("Value of existing key Mark: " + grades.get("Mark"));
    System.out.println("Value of existing key Tom: " + grades.get("Tom"));
    System.out.println("Value of non-existing key Simon: " + grades.get("Simon"));
    grades.put("Mark", "F");
    System.out.println("Value of existing key Mark: " + grades.get("Mark"));
    grades.remove("Alan");
    System.out.println("Key Alan exists: " + grades.containsKey("Alan"));
    System.out.println("Value of non-existing key Alan: " + grades.get("Alan"));
    }
}
```

Bad Example of using Generics

Has the following client made an appropriate choice?

```
Book<Object> book
```

NO**!!!!!!!!!!!!!!!!!!!!!!!**

- ◦ It allows **all** kinds of objects to be stored.
  ∵ All classes are descendants of ***Object***.
- ◦ We can expect **very little** from an object retrieved from this book.
  ∵ The ***static type*** of `book`'s items are ***Object***, root of the class hierarchy, has the <mark>*minimum*</mark> amount of methods available for use.
  ∵ Exhaustive list of casts are unavoidable.

  [***bad*** for extensibility and maintainability]

Beyond this lecture . . .

- Study `https://docs.oracle.com/javase/tutorial/java/generics/index.html` for further details on Java generics.
- Play with the source code `ExampleBooks`.
- Review the basic `ArrayList` and `HashTable` methods:
  - `ArrayList`:
    `https://www.youtube.com/watch?v=Gq_RRaGN7o8&list=PL5dxAmCmjv_4uhxBzBt-CnSGw6kZ9C-xe&index=5`
  - `Hashtable`:
    `https://www.youtube.com/watch?v=vM_JTnvDn1g&list=PL5dxAmCmjv_4uhxBzBt-CnSGw6kZ9C-xe&index=7`

Index (1)

Index (2)

Recursion

EECS2030 B & E: Advanced
Object Oriented Programming
Fall 2021

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

1. How to solve problems *recursively*

2. Example *recursions* on string and arrays

3. Some more advanced example (if time permitted)

Beyond this lecture ...

- Fantastic resources for sharpening your recursive skills for the exam:

  `http://codingbat.com/java/Recursion-1`

  `http://codingbat.com/java/Recursion-2`

- The *best* approach to learning about recursion is via a functional programming language:

  Haskell Tutorial: `https://www.haskell.org/tutorial/`

Recursion: Principle

- *Recursion* is useful in expressing solutions to problems that can be *recursively* defined:
  - *Base* **Cases:** Small problem instances immediately solvable.
  - *Recursive* **Cases:**
    - Large problem instances *not immediately solvable*.
    - Solve by reusing *solution(s) to strictly smaller problem instances*.
- Similar idea learnt in high school: [*mathematical induction*]
- Recursion can be easily expressed programmatically in Java:

```
m (i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m (j);/* recursive call with strictly smaller value */
  }
}
```

  - In the body of a method *m*, there might be *a call or calls to m itself*.
  - Each such self-call is said to be a *recursive call*.
  - Inside the execution of *m*(i), a recursive call *m*(j) must be that $j < i$.

Tracing Method Calls via a Stack

- When a method is called, it is ***activated*** (and becomes *active*) and ==*pushed*== onto the stack.

- When the body of a method makes a (helper) method call, that (helper) method is ***activated*** (and becomes *active*) and ==*pushed*== onto the stack.

  ⇒ The stack contains activation records of all *active* methods.
  - ==*Top*== of stack denotes the current point of execution.
  - Remaining parts of stack are (temporarily) ***suspended***.

- When entire body of a method is executed, stack is ==*popped*==.

  ⇒ The current point of execution is returned to the new ==*top*== of stack (which was ***suspended*** and just became ***active***).

- Execution terminates when the stack becomes ==*empty*==.

Recursion: Factorial (1)

• Recall the formal definition of calculating the *n* factorial:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

• How do you define the same problem *recursively*?

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1 \end{cases}$$

• To solve *n*!, we combine *n* and the solution to (*n - 1*)!.

```
int factorial (int n) {
  int result;
  if(n == 0) { /* base case */ result = 1; }
  else { /* recursive case */
    result = n * factorial (n - 1);
  }
  return result;
}
```

Common Errors of Recursive Methods

- Missing Base Case(s).

```
int factorial (int n) {
  return n * factorial (n - 1);
}
```

Base case(s) are meant as points of stopping growing the runtime stack.
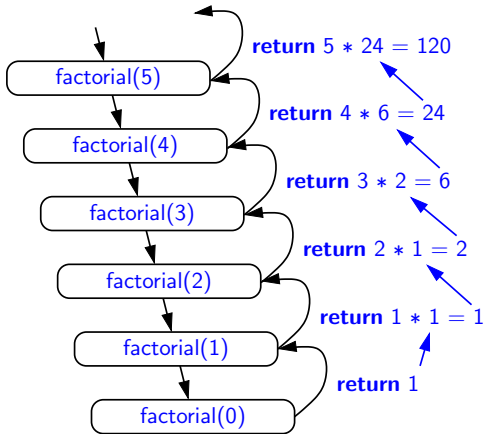
- Recursive Calls on Non-Smaller Problem Instances.

```
int factorial (int n) {
  if(n == 0) { /* base case */ return 1; }
  else { /* recursive case */ return n * factorial (n); }
}
```

Recursive calls on **strictly smaller** problem instances are meant for moving gradually towards the base case(s).

- In both cases, a StackOverflowException will be thrown.

Recursion: Factorial (3)

- When running *factorial(5)*, a *recursive call factorial(4)* is made. Call to *factorial(5)* suspended until *factorial(4)* returns a value.
- When running *factorial(4)*, a *recursive call factorial(3)* is made. Call to *factorial(4)* suspended until *factorial(3)* returns a value.

  . . .
- *factorial(0)* returns 1 back to *suspended call factorial(1)*.
- *factorial(1)* receives 1 from *factorial(0)*, multiplies 1 to it, and returns 1 back to the *suspended call factorial(2)*.
- *factorial(2)* receives 1 from *factorial(1)*, multiplies 2 to it, and returns 2 back to the *suspended call factorial(3)*.
- *factorial(3)* receives 2 from *factorial(1)*, multiplies 3 to it, and returns 6 back to the *suspended call factorial(4)*.
- *factorial(4)* receives 6 from *factorial(3)*, multiplies 4 to it, and returns 24 back to the *suspended call factorial(5)*.
- *factorial(5)* receives 24 from *factorial(4)*, multiplies 5 to it, and returns 120 as the result.

Recursion: Factorial (4)

- When the execution of a method (e.g., *factorial(5)*) leads to a nested method call (e.g., *factorial(4)*):
  - The execution of the current method (i.e., *factorial(5)*) is *suspended*, and a structure known as an **activation record** or **activation frame** is created to store information about the progress of that method (e.g., values of parameters and local variables).
  - The nested methods (e.g., *factorial(4)*) may call other nested methods (*factorial(3)*).
  - When all nested methods complete, the activation frame of the *latest suspended* method is re-activated, then continue its execution.
- What kind of data structure does this activation-suspension process correspond to?   [LIFO Stack]

Recursion: Fibonacci Sequence (1)

LASSONDE

- Can you identify the pattern of a Fibonacci sequence?

$$F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

- Here is the formal, **recursive** definition of calculating the n_{th} number in a Fibonacci sequence (denoted as F_n):

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
int  fib (int n) {
  int result;
  if(n == 1) { /* base case */ result = 1; }
  else if(n == 2) { /* base case */ result = 1; }
  else { /* recursive case */
    result = fib (n - 1) + fib (n - 2);
  }
  return result;
}
```

Recursion: Fibonacci Sequence (2)

```
      fib(5)
 =    {fib(5) = fib(4) + fib(3); push(fib(5)); suspended: ⟨fib(5)⟩; active: fib(4)}
      fib(4) + fib(3)
 =    {fib(4) = fib(3) + fib(2); suspended: ⟨fib(4), fib(5)⟩; active: fib(3)}
      ( fib(3) + fib(2)) + fib(3)
 =    {fib(3) = fib(2) + fib(1); suspended: ⟨fib(3), fib(4), fib(5)⟩; active: fib(2)}
      (( fib(2) + fib(1)) + fib(2)) + fib(3)
 =    {fib(2) returns 1; suspended: ⟨fib(3), fib(4), fib(5)⟩; active: fib(1)}
      (( 1 + fib(1) ) + fib(2)) + fib(3)
 =    {fib(1) returns 1; suspended: ⟨fib(3), fib(4), fib(5)⟩; active: fib(3)}
      (( 1 + 1 ) + fib(2)) + fib(3)
 =    {fib(3) returns 1 + 1; pop(); suspended: ⟨fib(4), fib(5)⟩; active: fib(2)}
      (2 + fib(2) ) + fib(3)
 =    {fib(2) returns 1; suspended: ⟨fib(4), fib(5)⟩; active: fib(4)}
      (2 + 1) + fib(3)
 =    {fib(4) returns 2 + 1; pop(); suspended: ⟨fib(5)⟩; active: fib(3)}
      3 + fib(3)
 =    {fib(3) = fib(2) + fib(1); suspended: ⟨fib(3),fib(5)⟩; active: fib(2)}
      3 + ( fib(2) + fib(1))
 =    {fib(2) returns 1; suspended: ⟨fib(3), fib(5)⟩; active: fib(1)}
      3 + (1 + fib(1) )
 =    {fib(1) returns 1; suspended: ⟨fib(3), fib(5)⟩; active: fib(3)}
      3 + (1 + 1)
 =    {fib(3) returns 1 + 1; pop() ; suspended: ⟨fib(5)⟩; active: fib(5)}
      3 + 2
      {fib(5) returns 3 + 2; suspended: ⟨⟩}
      5
```

Java Library: String

```java
public class StringTester {
  public static void main(String[] args) {
    String s = "abcd";
    System.out.println(s.isEmpty()); /* false */
    /* Characters in index range [0, 0) */
    String t0 = s.substring(0, 0);
    System.out.println(t0); /* "" */
    /* Characters in index range [0, 4) */
    String t1 = s.substring(0, 4);
    System.out.println(t1); /* "abcd" */
    /* Characters in index range [1, 3) */
    String t2 = s.substring(1, 3);
    System.out.println(t2); /* "bc" */
    String t3 = s.substring(0, 2) + s.substring(2, 4);
    System.out.println(s.equals(t3)); /* true */
    for(int i = 0; i < s.length(); i ++) {
      System.out.print(s.charAt(i));
    }
    System.out.println();
  }
}
```

Recursion: Palindrome (1)

Problem: A palindrome is a word that reads the same forwards and backwards. Write a method that takes a string and determines whether or not it is a palindrome.

```
System.out.println(isPalindrome(""));    true
System.out.println(isPalindrome("a"));    true
System.out.println(isPalindrome("madam"));    true
System.out.println(isPalindrome("racecar"));    true
System.out.println(isPalindrome("man"));    false
```

Base Case 1: Empty string ⟶ Return *true* immediately.

Base Case 2: String of length 1 ⟶ Return *true* immediately.

Recursive Case: String of length ≥ 2 ⟶
- 1st and last characters match, **and**
- *the rest (i.e., middle) of the string* is a palindrome .

Recursion: Palindrome (2)

```java
boolean isPalindrome (String word) {
  if(word.length() == 0 || word.length() == 1) {
    /* base case */
    return true;
  }
  else {
    /* recursive case */
    char firstChar = word.charAt(0);
    char lastChar = word.charAt(word.length() - 1);
    String middle = word.substring(1, word.length() - 1);
    return
        firstChar == lastChar
        /* See the API of java.lang.String.substring. */
        && isPalindrome (middle);
  }
}
```

Recursion: Reverse of String (1)

Problem: The reverse of a string is written backwards. Write a method that takes a string and returns its reverse.

```
System.out.println(reverseOf(""));   /* "" */
System.out.println(reverseOf("a"));   "a"
System.out.println(reverseOf("ab"));   "ba"
System.out.println(reverseOf("abc"));   "cba"
System.out.println(reverseof("abcd"));   "dcba"
```

Base Case 1: Empty string $\longrightarrow$ Return *empty string*.

Base Case 2: String of length 1 $\longrightarrow$ Return *that string*.

Recursive Case: String of length ≥ 2 $\longrightarrow$

   1) Head of string (i.e., first character)
   2) Reverse of the <u>tail of string</u> (i.e., all but the first character)

Return the concatenation of **2)** and **1)**.

Recursion: Reverse of a String (2)

```
String reverseOf (String s) {
  if(s.isEmpty()) { /* base case 1 */
    return "";
  }
  else if(s.length() == 1) { /* base case 2 */
    return s;
  }
  else { /* recursive case */
    String tail = s.substring(1, s.length());
    String reverseOfTail = reverseOf (tail);
    char head = s.charAt(0);
    return reverseOfTail + head;
  }
}
```

Recursion: Number of Occurrences (1)

Problem: Write a method that takes a string s and a character c, then count the number of occurrences of c in s.

```
System.out.println(occurrencesOf("", 'a')); /* 0 */
System.out.println(occurrencesOf("a", 'a')); /* 1 */
System.out.println(occurrencesOf("b", 'a')); /* 0 */
System.out.println(occurrencesOf("baaba", 'a')); /* 3 */
System.out.println(occurrencesOf("baaba", 'b')); /* 2 */
System.out.println(occurrencesOf("baaba", 'c')); /* 0 */
```

Base Case: Empty string $\longrightarrow$ Return *0*.

Recursive Case: String of length ≥ 1 $\longrightarrow$

1) Head of s (i.e., first character)

2) Number of occurrences of c in the <u>tail of s</u> (i.e., all but the first character)

If head is equal to c, return 1 + **2)**.

If head is not equal to c, return 0 + **2)**.

Recursion: Number of Occurrences (2)

```java
int  occurrencesOf (String s, char c) {
 if(s.isEmpty()) {
   /* Base Case */
   return 0;
 }
 else {
   /* Recursive Case */
   char head = s.charAt(0);
   String tail = s.substring(1, s.length());
   if(head == c) {
     return 1 +  occurrencesOf (tail, c);
   }
   else {
     return 0 +  occurrencesOf (tail, c);
   }
 }
}
```

Making Recursive Calls on an Array

- Recursive calls denote solutions to *smaller* sub-problems.
- *Naively*, explicitly create a new, smaller array:

```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
    m(sub) } }
```

- For *efficiency*, we pass the **reference** of the same array and specify the *range of indices* to be considered:

```
void m(int[] a, int from, int to) {
  if(from > to) { /* base case */ }
  else if(from == to) { /* base case */ }
  else { m(a, from + 1, to) } }
```

- m(a, 0, a.length - 1)   [Initial call; entire array]
- m(a, 1, a.length - 1)   [1st r.c. on array of size *a.length* – 1]
- m(a, a.length-1, a.length-1)   [Last r.c. on array of size 1]

Recursion: All Positive (1)

Problem: Determine if an array of integers are all positive.

```
System.out.println(allPositive({}));  /* true */
System.out.println(allPositive({1, 2, 3, 4, 5}));  /* true */
System.out.println(allPositive({1, 2, -3, 4, 5}));  /* false */
```

Base Case: Empty array ⟶ Return *true* immediately.
The base case is *true* ∵ we can *not* find a counter-example
(i.e., a number *not* positive) from an empty array.
Recursive Case: Non-Empty array ⟶
- 1st element positive, **and**
- *the rest of the array* is all positive .

Exercise: Write a method `boolean somePostive(int[] a)` which *recursively* returns *true* if there is some positive number in `a`, and *false* if there are no positive numbers in `a`.
Hint: What to return in the base case of an empty array? [*false*]
∵ No witness (i.e., a positive number) from an empty array

Recursion: All Positive (2)

```
boolean allPositive(int[] a) {
  return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return a[from] > 0;
  }
  else { /* recursive case */
    return a[from] > 0 && allPositiveHelper(a, from + 1, to);
  }
}
```

Problem: Determine if an array of integers are sorted in a non-descending order.

```
System.out.println(isSorted({}));   true

System.out.println(isSorted({1, 2, 2, 3, 4}));   true

System.out.println(isSorted({1, 2, 2, 1, 3}));   false
```

Base Case: Empty array ⟶ Return *true* immediately.
The base case is *true* ∵ we can *not* find a counter-example
(i.e., a pair of adjacent numbers that are *not* sorted in a
non-descending order) from an empty array.
Recursive Case: Non-Empty array ⟶
- 1st and 2nd elements are sorted in a non-descending order, **and**
- *the rest of the array*, starting from the 2nd element,
  are sorted in a non-descending order .

Recursion: Is an Array Sorted? (2)

```java
boolean isSorted(int[] a) {
  return isSortedHelper(a, 0, a.length - 1);
}

boolean isSortedHelper(int[] a, int from, int to) {
  if (from > to) { /* base case 1: empty range */
    return true;
  }
  else if(from == to) { /* base case 2: range of one element */
    return true;
  }
  else {
    return a[from] <= a[from + 1]
      && isSortedHelper(a, from + 1, to);
  }
}
```

Beyond this lecture ...

- Recursions on Arrays: Lab Exercise from EECS2030-F19
- Notes on Recursion:

  http://www.eecs.yorku.ca/~jackie/teaching/
  lectures/2021/F/EECS2030/notes/EECS2030_F21
  Notes_Recursion.pdf

- API for String:

  https://docs.oracle.com/javase/8/docs/api/
  java/lang/String.html

- Fantastic resources for sharpening your recursive skills for the exam:

  http://codingbat.com/java/Recursion-1
  http://codingbat.com/java/Recursion-2

- The *best* approach to learning about recursion is via a functional programming language:
  Haskell Tutorial: https://www.haskell.org/tutorial/

Index (1)