Classes and Objects

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

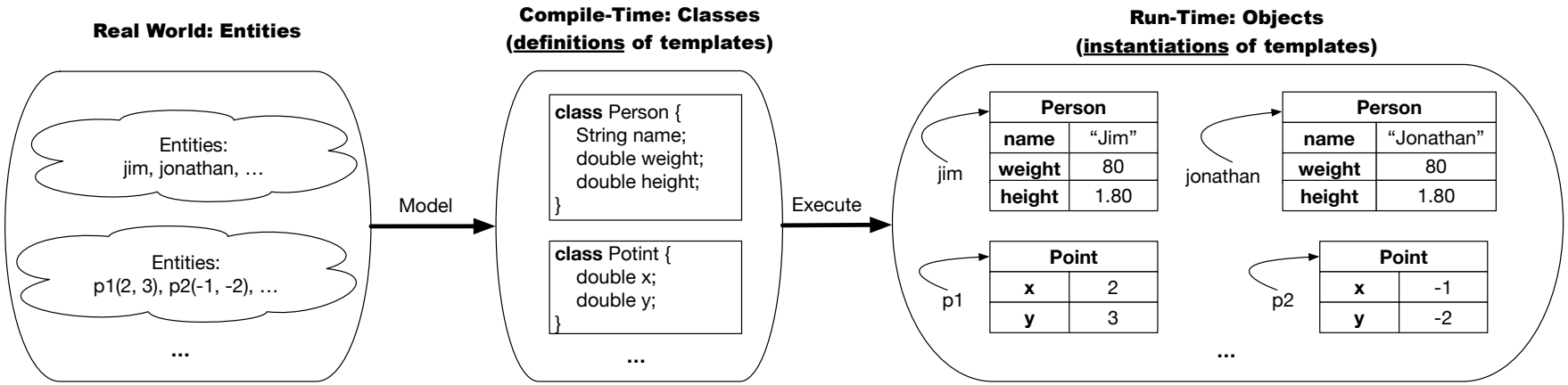
Separation of Concerns: App/Tester vs. Model

- In EECS1022:
  - ***Model Component***: One or More Java Classes
    e.g., `Person` vs. `SMS`, `Student`, `CourseRecord`
  - Another Java class that "manipulates" the model class
    (by creating instances and calling methods):
    - ***Controller*** (e.g., `BMIActivity`, `BankActivity`). Effects?
      Visualized (via a GUI) at connected tablet
    - ***Tester*** with `main` (e.g., `PersonTester`, `BankTester`). Effects?
      Seen (as textual outputs) at console
- In Java:
  - We may define more than one *classes*.
  - Each class may contain more than one *methods*.

object-oriented programming 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



- 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.

OO Thinking: Templates vs. Instances (1.1)

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 (1.2)

- Persons share these common *attributes* and *behaviour*.
  - Each person possesses an age, a weight, and a height.
  - Each person's age, weight, and height might be *distinct*
    e.g., `jim` is 50-years old, 1.8-meters tall and 80-kg heavy
    e.g., `jonathan` is 65-years old, 1.73-meters tall and 90-kg heavy
- Each person, depending on the *specific values* of their attributes, might 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 $[\frac{90}{1.73^2}]$

OO Thinking: Templates vs. Instances (2.1)

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 (2.2)

- Points share these common *attributes* and *behaviour*.
  - Each point possesses an x-coordinate and a y-coordinate.
  - Each point's location might be *distinct*
    e.g., `p1` is located at $(3, 4)$
    e.g., `p2` is located at $(-4, -3)$
- Each point, depending on the *specific values* of their attributes (i.e., locations), might 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: $[\sqrt{(-4)^2 + (-2)^2}]$

OO Thinking: Templates vs. Instances (3)

- A *template* defines what's **shared** by a set of related entities.
  - Common *attributes* (`age` in `Person`, `x` in `Point`)
  - Common *behaviour* (get older for `Person`, move up for `Point`)
- Each template may be *instantiated* into multiple instances.
  - `Person` instances: `jim` and `jonathan`
  - `Point` instances: `p1` and `p2`
- Each *instance* may have *specific values* for the attributes.
  - Each `Person` instance has an age:
    `jim` is 50-years old, `jonathan` is 65-years old
  - Each `Point` instance has a location:
    `p1` is at $(3, 4)$, `p2` is at $(-3, -4)$
- Therefore, instances of the same template may exhibit *distinct behaviour*.
  - Each `Person` instance can get older: `jim` getting older from 50 to 51; `jonathan` getting older from 65 to 66.
  - Each `Point` instance can move up: `p1` moving up from $(3, 3)$ results in $(3, 4)$; `p1` moving up from $(-3, -4)$ results in $(-3, -3)$.

OOP: Classes ≈ Templates

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

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

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

OOP:
Define Constructors for Creating Objects (1.1)

- Within class `Point`, you define *constructors*, specifying how instances of the `Point` template may be created.

```
public class Point {
  ... /* attributes: x, y */
  Point(double newX, double newY) {
    x = newX;
    y = newY; } }
```

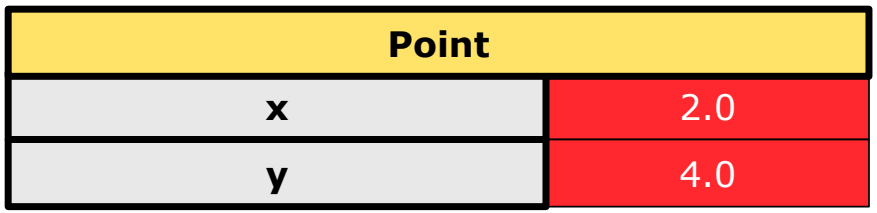
- In the corresponding tester class, each *call* to the `Point` constructor creates an instance of the `Point` template.

```
public class PointTester {
  public static void main(String[] args) {
    Point p1 = new Point(2, 4);
    println(p1.x + " " + p1.y);
    Point p2 = new Point(-4, -3);
    println(p2.x + " " + p2.y); } }
```

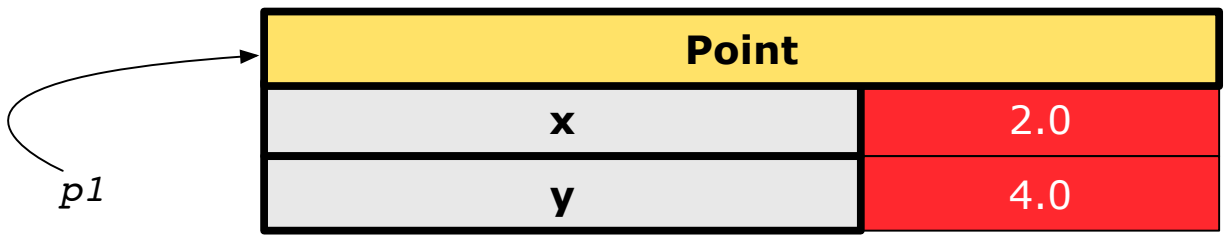
OOP:
Define Constructors for Creating Objects (1.2)

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

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



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`.



OOP:
Define Constructors for Creating Objects (2.1)

- Within class `Person`, you define *constructors*, specifying how instances of the `Person` template may be created.

```
public class Person {
  ... /* attributes: age, nationality, weight, height */
  Person(int newAge, String newNationality) {
    age = newAge;
    nationality = newNationality; } }
```

- In the corresponding tester class, each *call* to the `Person` constructor creates an instance of the `Person` template.

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

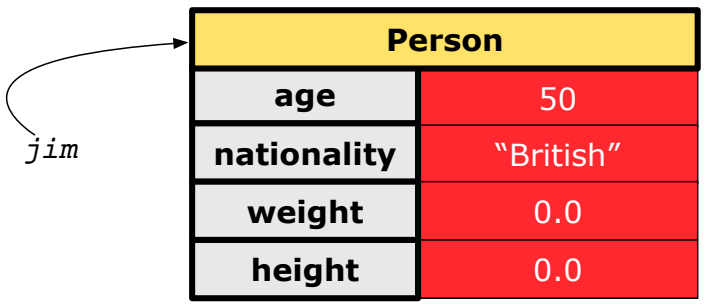
OOP:
Define Constructors for Creating Objects (2.2)

```
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`.



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*

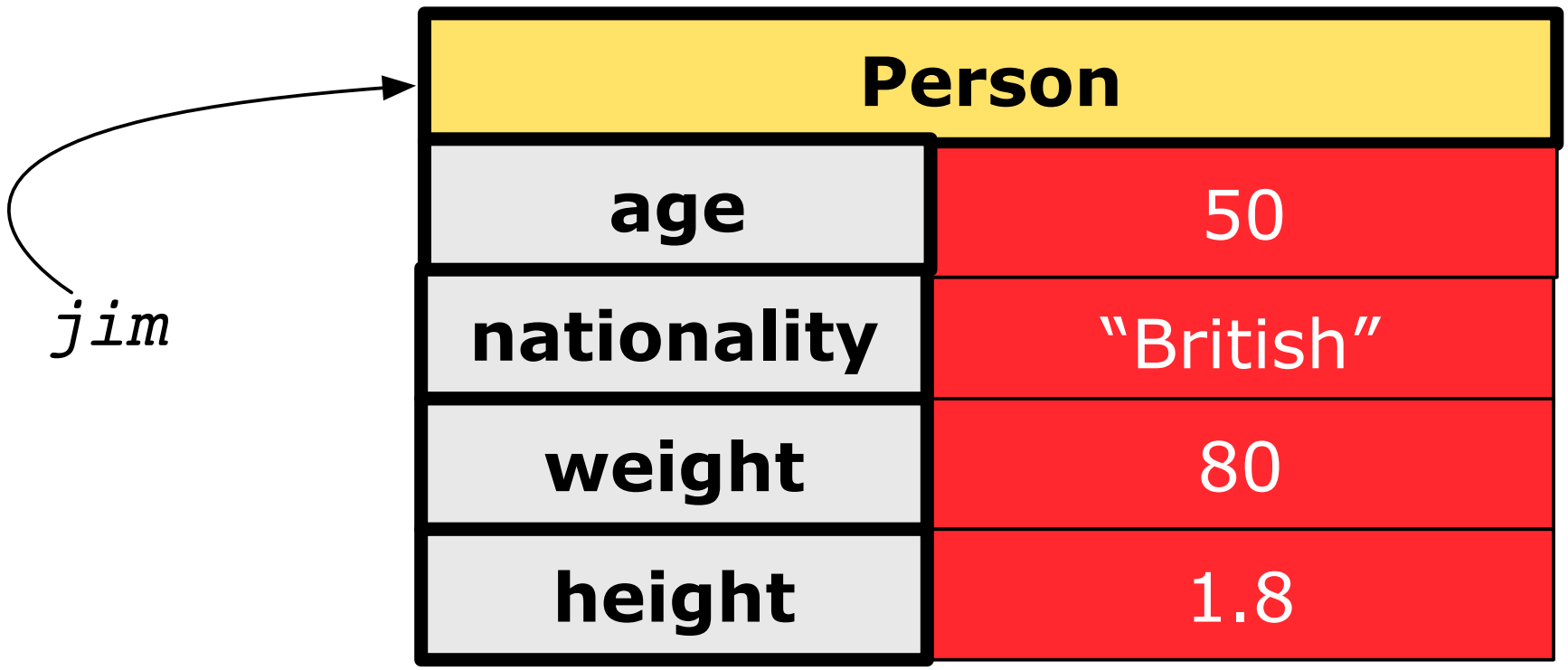
    ```
    jim.gainWeightBy(10);
    ```

- 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);
```

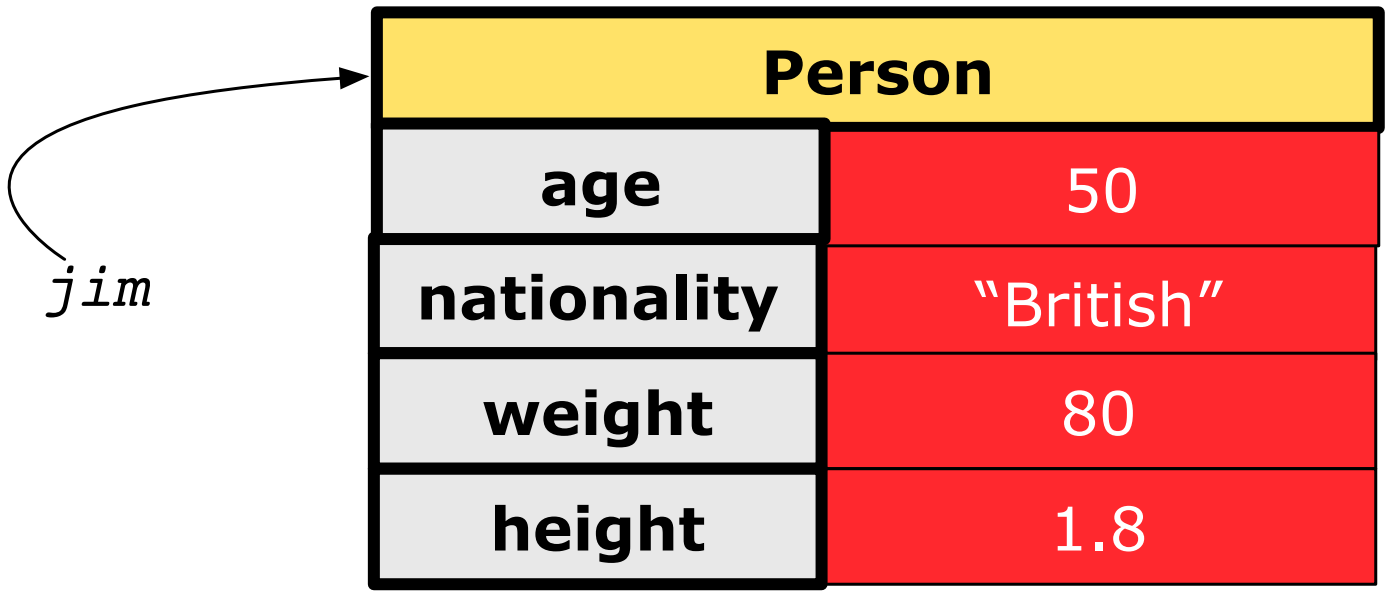


Visualizing Objects at Runtime (2.2)

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`.

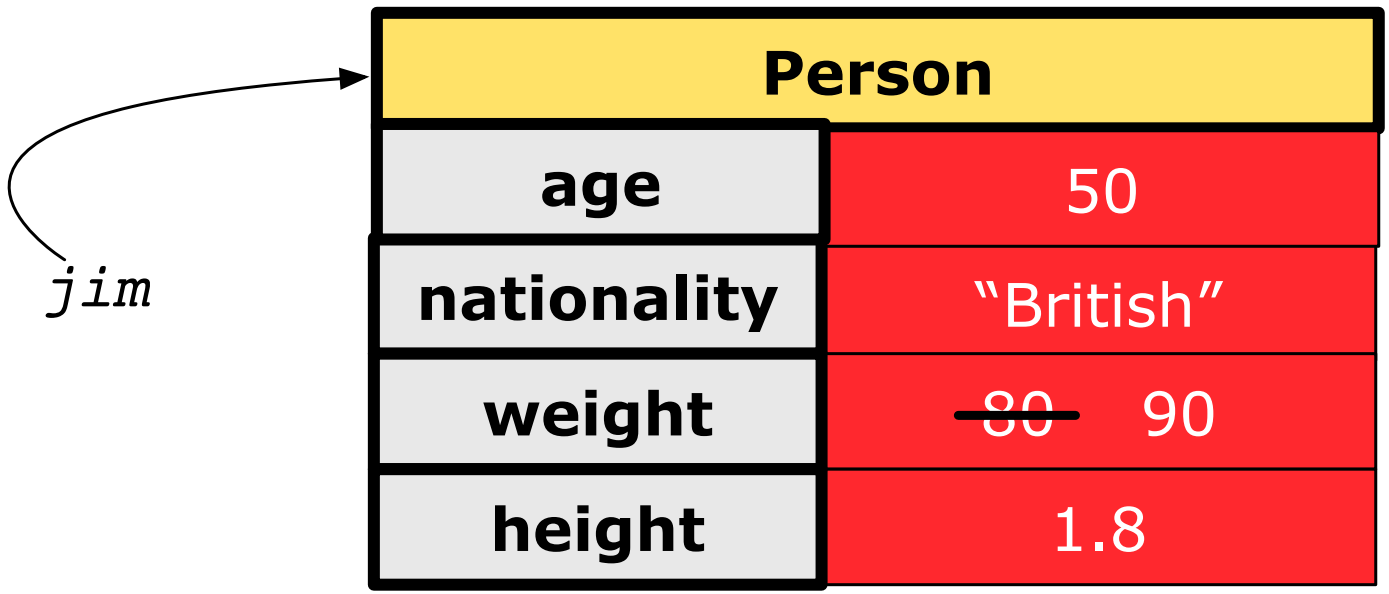


Visualizing Objects at Runtime (2.3)

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!

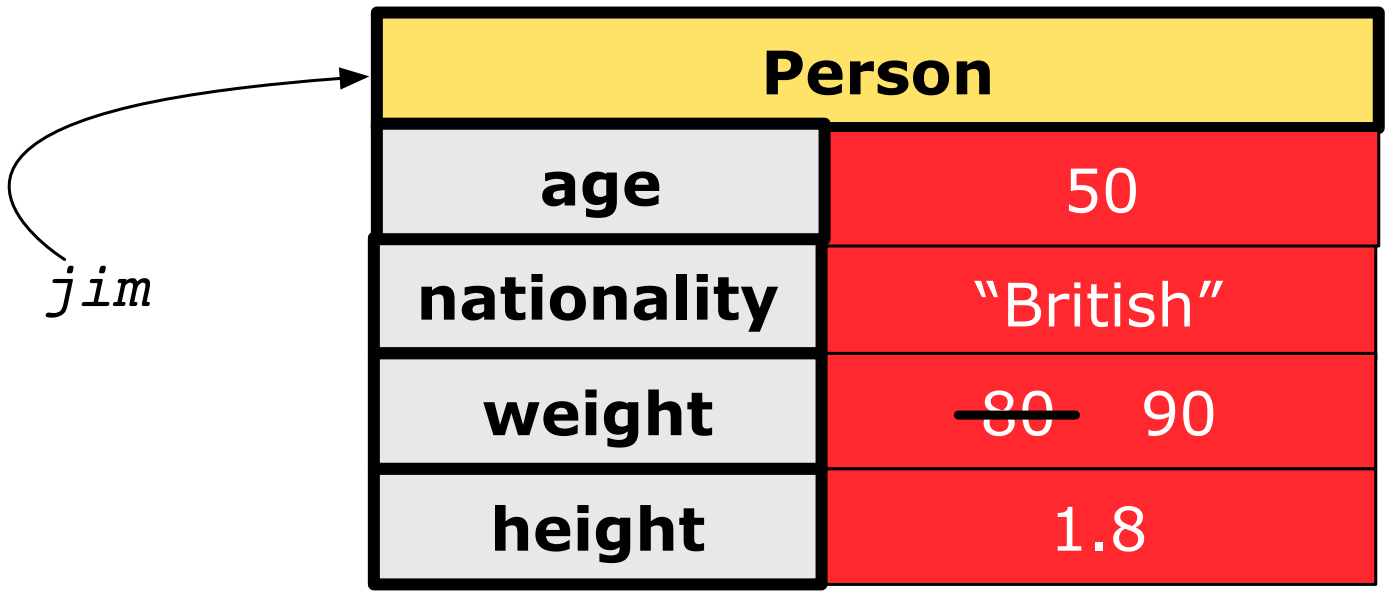


Visualizing Objects at Runtime (2.4)

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

```
bmi = p.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`.



The `this` Reference (1)

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

```
class Point {
  double x; double y;
  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 *context object* in a call to that method.

```
class Point {
  double x;
  double y;
  Point(double newX, double newY) {
    this.x = newX;
    this.y = newY;
  }
  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:

```
class Point {
  double x;
  double y;
  Point(double newX, double newY) {
    p1.x = newX;
    p1.y = newY;
  }
  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:

```
class Point {
  double x;
  double y;
  Point(double newX, double newY) {
    p2.x = newX;
    p2.y = newY;
  }
  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*.

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

The `this` Reference (6.1): Common Error

The following code fragment compiles but is problematic:

```
class Person {
  String name;
  int age;
  Person(String name, int age) {
    name = name;
    age = age;
  }
  void setAge(int age) {
    age = age;
  }
}
```

Why? Fix?

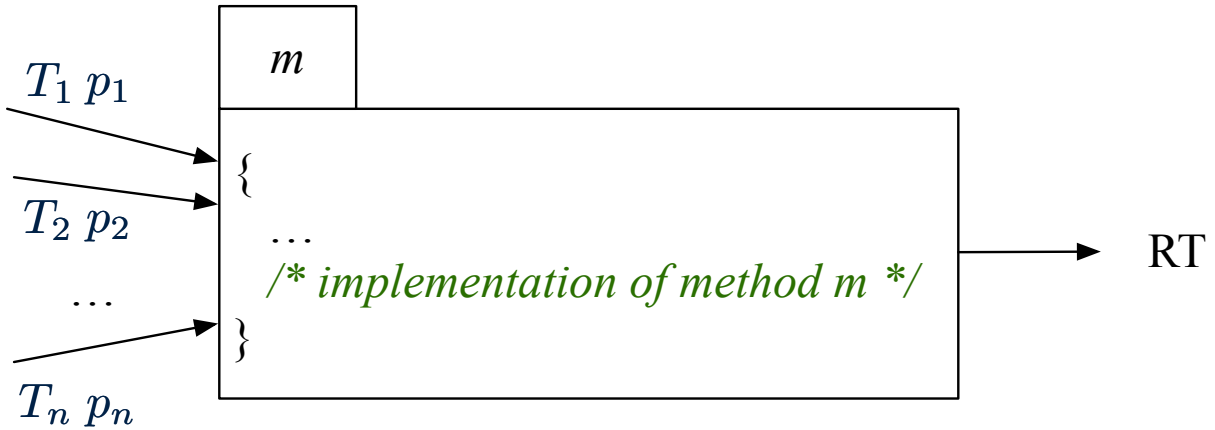
The `this` Reference (6.2): Common Error

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

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

OOP: Methods (1.1)

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



- The *header* of a method consists of: [see here]
  - 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 and use new *local variables*
    Scope of local variables is only within that method.
  - Use or change values of *attributes*.
  - Use values of *parameters*, if any.

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

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

```
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: The Dot Notation (1.1)

A binary operator:

- `LHS` stores an address (which denotes an object)
- `RHS` the name of an attribute or a method
- `LHS.RHS` means:

  Locate the context object whose address is stored in **LHS**, then apply `RHS`.

  What if `LHS` stores `null`? [`NullPointerException`]

OOP: The Dot Notation (1.2)

- 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()`

OOP: Method Calls

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

OOP: Class Constructors (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*

OOP: Class Constructors (2)

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

OOP: Class Constructors (3)

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

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

  Point(char axis, double distance) {
    if (axis == 'x') { x = distance; }
    else if (axis == 'y') { y = distance; }
    else { System.out.println("Error: invalid axis.") }
  }
}
```

OOP: Class Constructors (4)

- 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.
  - Each constructor must have a *distinct* list of *input parameter types*.
  - 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: Object Creation (1)

```
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.x + ", " + p1.y + ")");
```

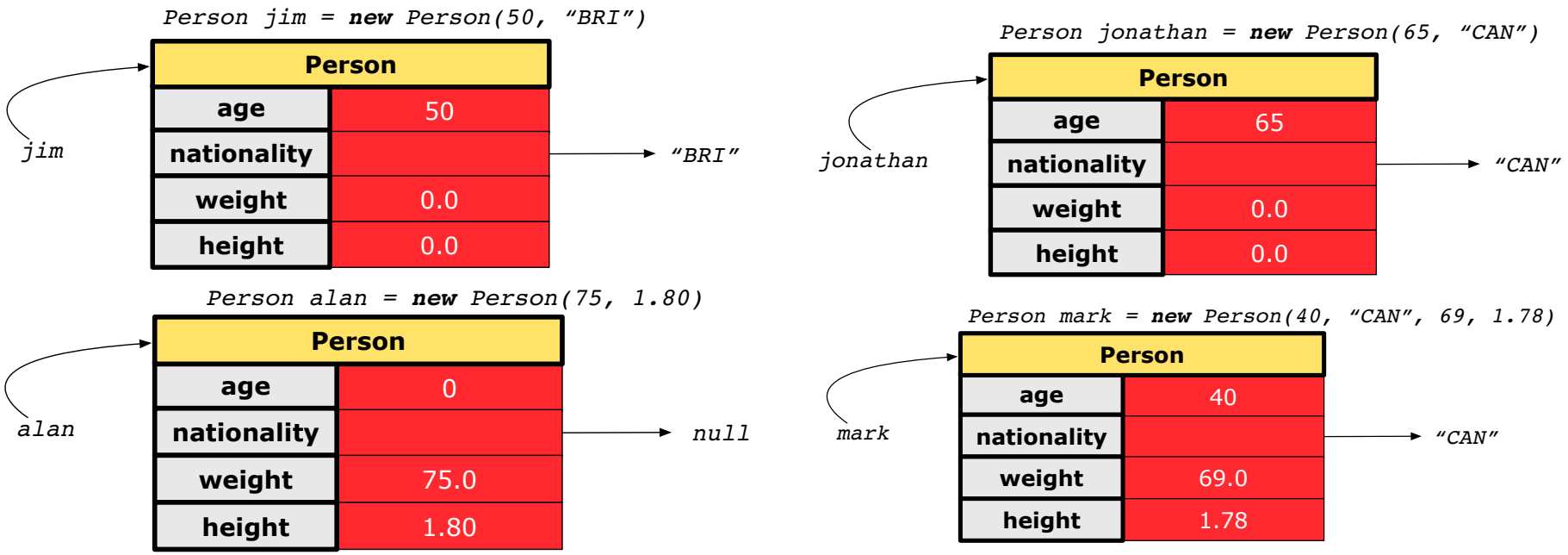
```
(2.0, 4.0)
```

OOP: Object Creation (2)

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

```
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)

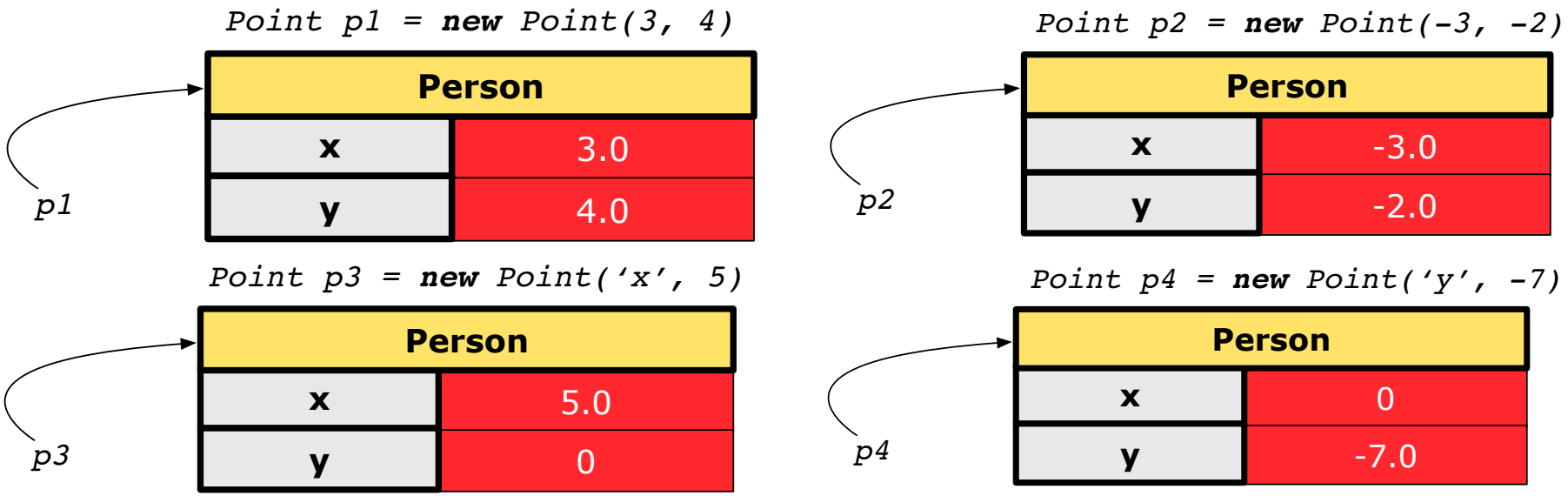


OOP: Object Creation (4)

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 (5)



OOP: Object Creation (6)

- 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: Mutator Methods

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

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

```
public class Point {
  ...
  void moveUp() {
    y = 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 {
  ...
  double getBMI() {
    double bmi = height / (weight * weight);
    return bmi;
  }
}
```

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

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: Object Alias (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.

OO Program Programming: Object Alias (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]);
```

OO Program Programming: Object Alias (2.2)

Problem: 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 + 1) % persons1.length]; }
9  persons1[0].setAge(70);
10 System.out.println(jim.age);  /* 0 */
11 System.out.println(alan.age);  /* 70 */
12 System.out.println(persons2[0].age);  /* 0 */
13 persons1[0] = jim;
14 persons1[0].setAge(75);
15 System.out.println(jim.age);  /* 75 */
16 System.out.println(alan.age);  /* 70 */
17 System.out.println(persons2[0].age);  /* 0 */
```

OO Program Programming: Object Alias (3)

```
Person tom = new Person("TomCruise");
Person ethanHunt = tom;
Person spy = ethanHunt;
tom.setWeight(77);  print(tom.weight);  /* 77 */
ethanHunt.gainWeight(10);  print(tom.weight);  /* 87 */
spy.loseWeight(10);  print(tom.weight);  /* 77 */
Person prof = new Person("Jackie"); prof.setWeight(80);
spy = prof;   prof = tom;   tom = spy;
print(prof.name+" teaches 2030");/*TomCruise teaches 2030*/
print("EthanHunt is "+ethanHunt.name);/*EthanHunt is TomCruise*/
print("EthanHunt is "+spy.name);/*EthanHunt is Jackie*/
print("TomCruise is "+tom.name);/*TomCruise is Jackie*/
print("Jackie is "+prof.name);/*Jackie is TomCruise*/
```

- An *object* at runtime may have *more than one identities*.
  Its *address* may be stored in multiple *reference variables*.
- Calling a *method* on one of an object's identities has the *same effect* as calling the same method on any of its other identities.

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 reused (without recalculating) via the name `sqr`.
- RHS: it is not stored 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)* denotes an object without a name reference.

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

Anonymous Objects (2.2)

One more example on using anonymous objects:

```
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) );
  }
}
```

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 primitive variable that is *uninitialized*.
  Make sure the *default value* is what you want!
- Calling a method on a *uninitialized* reference variable crashes your program. [*NullPointerException*]
  Always initialize reference variables!

Java Data Types (3.1)

- An attribute may store the reference to some object.

```
class Person { Person spouse; }
```

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

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

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

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

Java Data Types (3.2.1)

An attribute may be of type `Point[]`, storing references to `Point` objects.

```
1  class PointCollector {
2    Point[] points; int nop; /* number of points */
3    PointCollector() { points = new Point[100]; }
4    void addPoint(double x, double y) {
5      points[nop] = new Point(x, y); nop++; }
6    Point[] getPointsInQuadrantI() {
7      Point[] ps = new Point[nop];
8      int count = 0; /* number of points in Quadrant I */
9      for(int i = 0; i < nop; i ++) {
10       Point p = 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

Java Data Types (3.2.2)

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

Static Variables (1)

```
class Account {
  int id;
  String owner;
  Account(int id, String owner) {
    this.id = id;
    this.owner = owner;
  }
}
```

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

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

Static Variables (2)

```
class Account {
  static int globalCounter = 1;
  int id; String owner;
  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.id != acc2.id); }
```

- 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 share a *single* copy of the ***static*** variable.
  - Change to `globalCounter` via `c1` is also visible to `c2`.

Static Variables (3)

```
class Account {
  static int globalCounter = 1;
  int id; String owner;
  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

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

```
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.accounts[0] */
  steve.addAccount(acc2);
    /* mistakenly added to steve.accounts[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:

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

Static Variables (5.1): Common Error

```
1  public class Bank {
2      public string branchName;
3      public static int nextAccountNumber = 1;
4      public static void useAccountNumber() {
5          System.out.println (branchName + ...);
6          nextAccountNumber ++;
7      }
8  }
```

- *Non-static method cannot be referenced from a static context*
- **Line 4** declares that we *can* call the method `userAccountNumber` without instantiating an object of the class `Bank`.
- However, in **Lined 5**, 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      public string branchName;
3      public static int nextAccountNumber = 1;
4      public static void useAccountNumber() {
5          System.out.println (branchName + ...);
6          nextAccountNumber ++;
7      }
8  }
```

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

```
Bank.useAccountNumber();
```

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

```
Bank b1 = new Bank(); b1.setBranch("Songdo IBK");
System.out.println(b1.branchName);
```

Static Variables (5.3): Common Error

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 (6)

Static Variables (5.3): Common Error

Exceptions

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

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***.

Why Exceptions? (1.1)

```
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*** `setRadius` to stop and handle invalid values of `r`.

Why Exceptions? (1.2)

```
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 force `CircleCalculator.main`, *caller* of `Circle.setRadius`, to realize that things might go wrong.
  ⇒ When things do go wrong, immediate actions are needed.

Why Exceptions? (2.1)

```
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 force the *caller* of `Account.deposit` or `Account.withdraw` to stop and handle invalid values of `a`.

Why Exceptions? (2.2)

```
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 force ***Bank.withdrawFrom***, the *caller* of ***Account.withdraw***, to stop and handle invalid values of `a`.

Why Exceptions? (2.3)

```
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  }
```

- 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`***, printed from ***`Account.withdraw`*** to console.
  - Impossible to force ***`BankApplication.main`***, the *caller* of ***`Bank.withdrawFrom`***, to stop and handle invalid values of `a`.
- **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

Exceptions in Java (1.1)

```
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 *signature* that it may *throw* an `InvalidRaidusException` object.

Exceptions in Java (1.2)

```
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 *signature* 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* .

Exceptions in Java (1.3)

```
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**.

Exceptions in Java (1.4.1)

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

Exceptions in Java (1.4.2)

```
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***]

Exceptions in Java (2.1)

```
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 *signature* that it may *throw* an `InvalidTransactionException` object.

Exceptions in Java (2.2)

```
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 *signature* 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* .

Exceptions in Java (2.3)

```
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 *signature* 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.

Exceptions in Java (2.4)

```
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**.

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();
}
```

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.

Example (2.2)

```
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 */
  }
}
```

Example: to Handle or Not to Handle? (1.1)

Consider the following three classes:

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

```
class B {
  mb(int i) {
    A oa = new A();
    oa.ma(i); /* Error occurs if i < 0 */
  } }
```

```
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?  */
  } }
```

Example: to Handle or Not to Handle? (1.2)

- 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.

Example: to Handle or Not to Handle? (2.1)

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

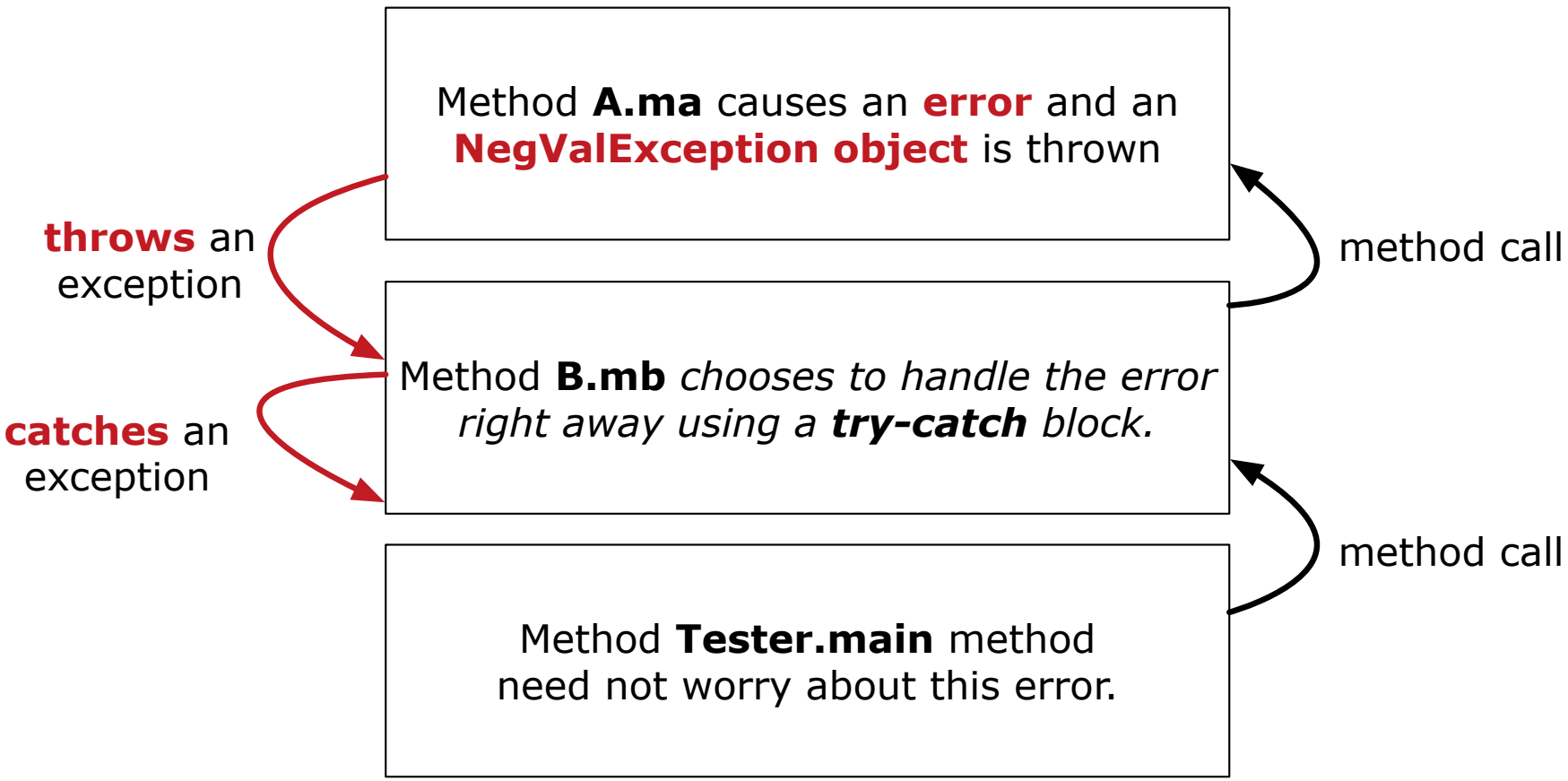
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

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

```
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. */
  } }
```

Example: to Handle or Not to Handle? (2.2)

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



Example: to Handle or Not to Handle? (3.1)

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

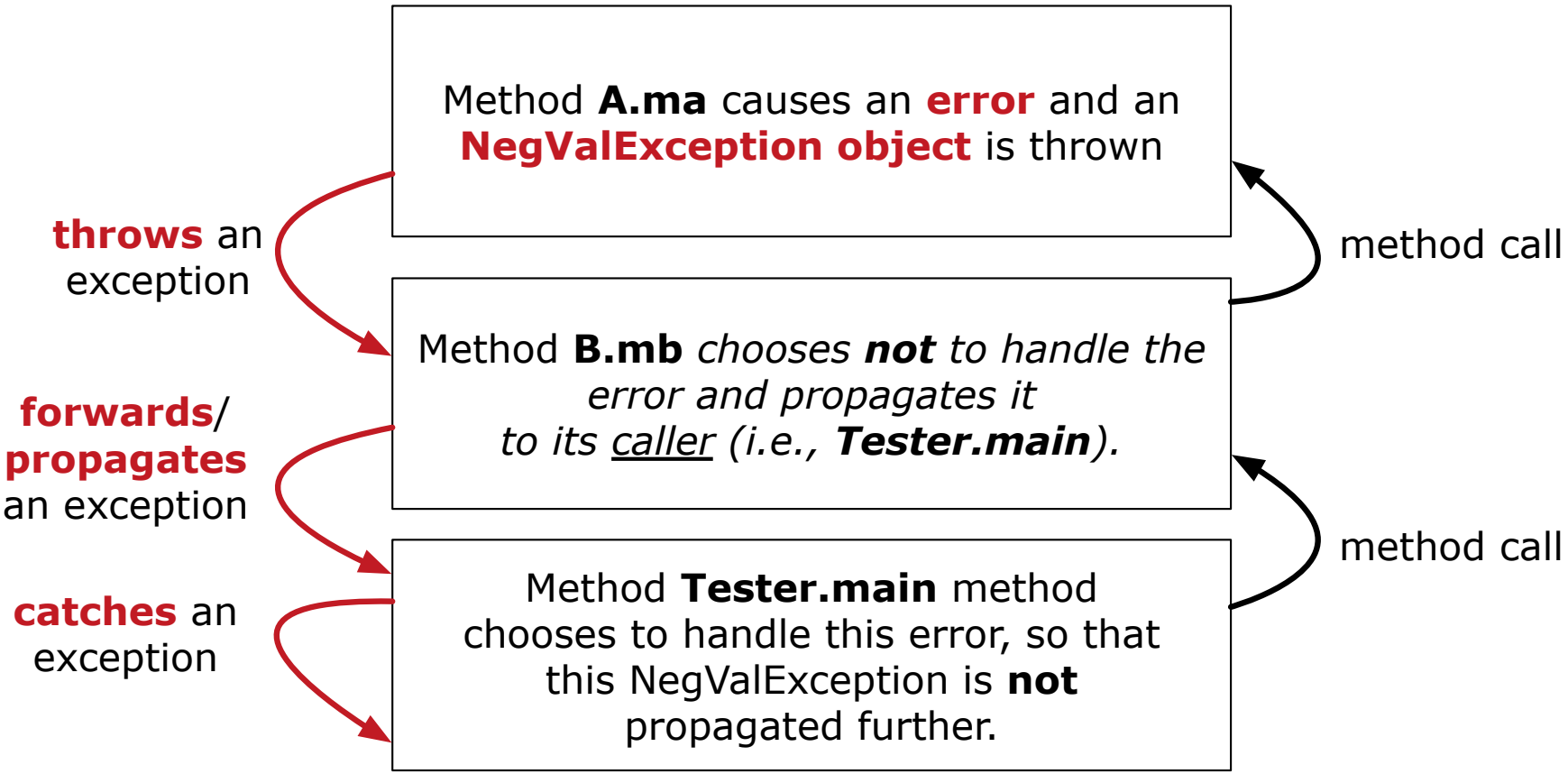
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
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. */ }
  } }
```

Example: to Handle or Not to Handle? (3.2)

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



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

Version 3: Handle in neither of the classes.

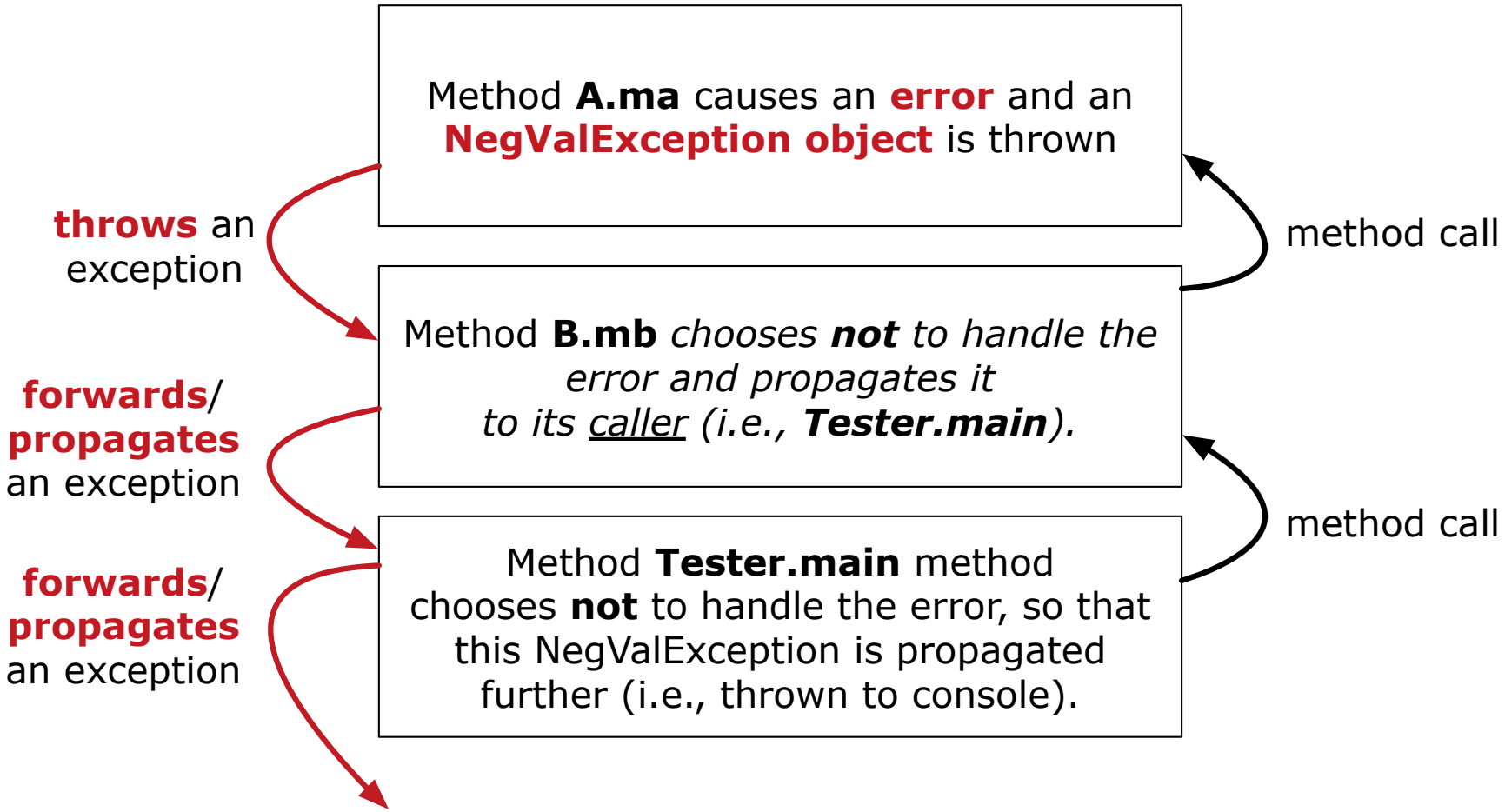
```
class A {
  ma(int i) throws NegValException {
    if(i < 0) { throw new NegValException("Error."); }
    else { /* Do something. */ }
  } }
```

```
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);
  } }
```

Example: to Handle or Not to Handle? (4.2)

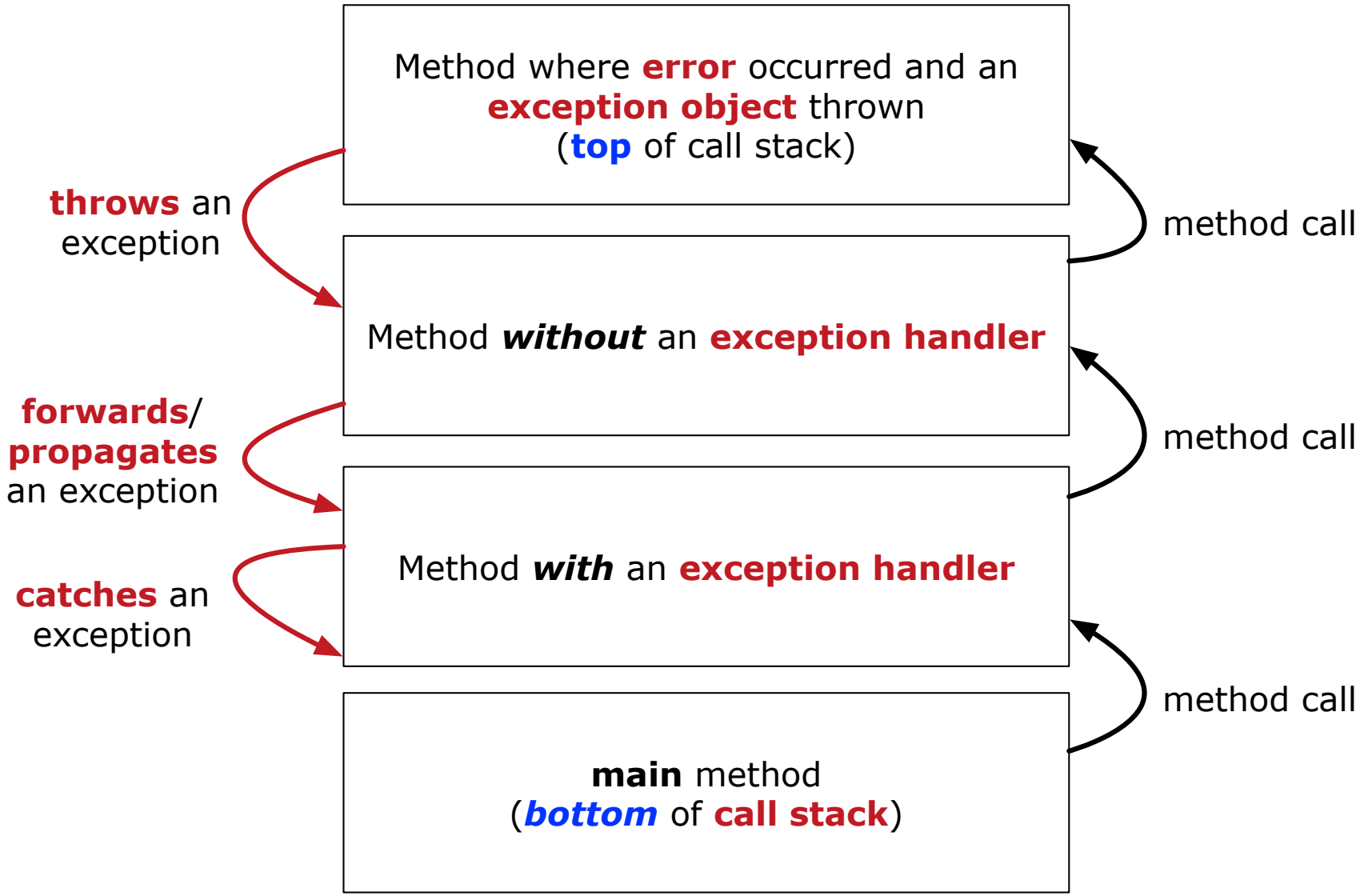
Version 3: Handle in neither of the classes.



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 call this chain of method calls 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.

What to Do When an Exception Is Thrown? (1)



What to Do When an Exception Is Thrown? (2)

- After a method *throws an exception*, the *runtime system* searches the corresponding *call stack* for a method that contains a block of code to *handle* the exception.
  - This block of code is called an *exception handler*.
    - 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 *call stack* without finding an **appropriate** *exception handler*
    ⇒ The program terminates and the exception object is directly "thrown" to the console!

The Catch or Specify Requirement (1)

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*.

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

The Catch or Specify Requirement (2)

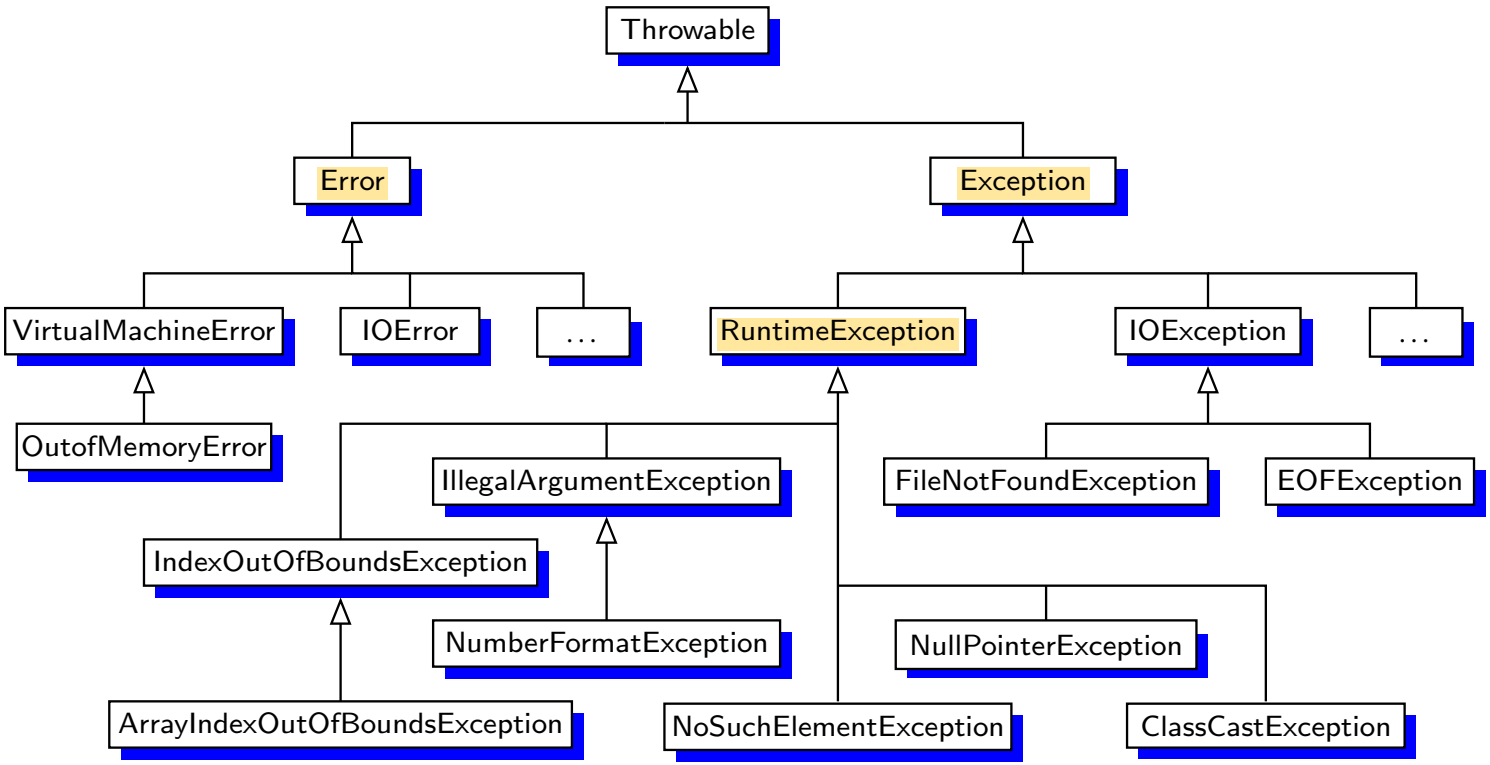
Code (e.g., a method call) that might throw certain exceptions must be enclosed by one of the two ways:

2. **The "Specify" Solution:** A method that specifies as part of its *signature* that it *can throw* the exception (without handling that exception).

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

The Catch or Specify Requirement (3)

There are *three* basic categories of exceptions



Only one category of exceptions is subject to the *Catch or Specify Requirement*.

Exception Category (1): Checked Exceptions

- *Checked exceptions* are exceptional conditions that a well-written application should anticipate and recover from.
  - An application prompts a user for a circle radius, a deposit/withdraw amount, or the name of a file to open.
  - *Normally*, the user enters a positive number for radius/deposit, a not-too-big positive number for withdraw, and existing file to open.
  - When the user enters invalid numbers or file names, `NegativeRadiusException`, `InvalidTransactionException`, or `FileNotFoundException` is thrown.
  - A well-written program will *catch* this exception and notify the user of the mistake.
- *Checked exceptions* are:
  - subject to the *Catch or Specify Requirement*.
  - subclasses of `Exception` that are **not** *descendant classes* of `RuntimeException`.

Exception Category (2): Errors

- *Errors* are exceptional conditions that are *external* to the application, and that the application usually cannot anticipate or recover from.
  - An application successfully opens a file for input.
  - But the file cannot be read because of a hardware or system malfunction.
  - The unsuccessful read will throw `java.io.IOError`
- *Errors* are:
  - *not* subject to the *Catch or Specify Requirement*.
  - subclasses of `Error`

Exception Category (3): Runtime Exceptions

- *Runtime exceptions* are exceptional conditions that are *internal* to the application, and that the application usually cannot anticipate or recover from.
  - These usually indicate programming bugs, such as logic errors or improper use of an API.

  e.g., `NullPointerException`

  e.g., `ClassCastException`

  e.g., `ArrayIndexOutOfBoundException`
- *Runtime exceptions* are:
  - *not* subject to the *Catch or Specify Requirement*.
  - subclasses of `RuntimeException`
- *Errors* and *Runtime exceptions* are collectively known as *unchecked exceptions*.

Catching and Handling Exceptions

- To construct an *exception handler*:
  1. Enclose the code that might throw an exception within a `try` block.
  2. Associate *each possible kind of exception* that might occur within the `try` block with a `catch` block.
  3. Append an optional `finally` block.

```
try { /* code that might throw exceptions */ }
catch(ExceptionType1 e) { ... }
catch(ExceptionType2 e) { ... }
...
finally { ... }
```

- When an exception is thrown from Line *i* in the *`try`* block:
  - Normal flow of execution is *interrupted*: the rest of *`try`* block starting from Line *i* + 1 is skipped.
  - Each *`catch`* block performs an `instanceof` check on the thrown exception: the first matched *`catch`* block is executed.
  - The *`finally`* block is always executed after the matched `catch` block is executed.

Examples (3)

```
double r = ...;
double a = ...;
try{
  Bank b = new Bank();
  b.addAccount(new Account(34));
  b.deposit(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();
}
catch( Exception e) { /* any other kinds of exceptions */
  e.printStackTrace();
}
```

Examples (4): Problem?

```
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());
}
/* Every exception object is a descendant of Exception.  */
catch( Exception e) {
  e.printStackTrace();
}
catch(NegativeRadiusException e) { /* Problem:  Not reachable!  */
  System.out.println(r + " is not a valid radius value.");
  e.printStackTrace();
}
catch(InvalidTransactionException e) { /* Problem:  Not reachable!  */
  System.out.println(r + " is not a valid transaction value.");
  e.printStackTrace();
}
```

Index (1)

Index (2)

Index (3)

Test-Driven Development (TDD) with JUnit

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Motivating Example: Two Types of Errors (1)

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);
  }
}
```

Motivating Example: Two Types of Errors (3)

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:

```
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.

Exceptional Scenarios

Consider the two possible exceptional scenarios:

- An attempt to increment ***above*** the counter's upper bound.
- An attempt to decrement ***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("counter value is " + value);
    }
    else { value ++; }
  }

  public void decrement() throws ValueTooSmallException {
    if(value == Counter.MIN_VALUE) {
      throw new ValueTooSmallException("counter 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()`.
- What do you *expect to happen*?

  e.g., value of counter is such that `Counter.MIN_VALUE + 1`
- What does your program *actually produce*?

  e.g., call `c.getValue` to find out.
- A test:
  - ***Passes*** if expected value *matches* actual value
  - ***Fails*** if expected value *does not match* actual value
- So far, you ran tests via a tester class with the `main` method.

Testing Counter from Console (V1): Case 1

Consider a class for testing the `Counter` class:

```
public class CounterTester1 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Init val: " + c.getValue());
    try {
      c.decrement();
      println("ValueTooSmallException NOT thrown as expected.");
    }
    catch (ValueTooSmallException e) {
      println("ValueTooSmallException thrown as expected.");
    } } }
```

Executing it as Java Application gives this Console Output:

```
Init val: 0
ValueTooSmallException thrown as expected.
```

Testing Counter from Console (V1): Case 2

Consider **another** class for testing the `Counter` class:

```
public class CounterTester2 {
  public static void main(String[] args) {
    Counter c = new Counter();
    println("Current val: " + c.getValue());
    try { c.increment(); c.increment(); c.increment(); }
    catch (ValueTooLargeException e) {
      println("ValueTooLargeException thrown unexpectedly."); }
    println("Current val: " + c.getValue());
    try {
      c.increment();
      println("ValueTooLargeException NOT thrown as expected."); }
    catch (ValueTooLargeException e) {
      println("ValueTooLargeException thrown as expected."); } } }
```

Executing it as Java Application gives this Console Output:

```
Current val: 0
Current val: 3
ValueTooLargeException thrown as expected.
```

Testing Counter from Console (V2)

Consider a different class for testing the `Counter` class:

```
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!"); }
      }
      catch(ValueTooLargeException e){ println("Value too big!"); }
      catch(ValueTooSmallException e){ println("Value too small!"); }
    } } }
```

Testing Counter from Console (V2): Test 1

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!
```

Testing Counter from Console (V2): Test 2

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 inconvenient 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**.

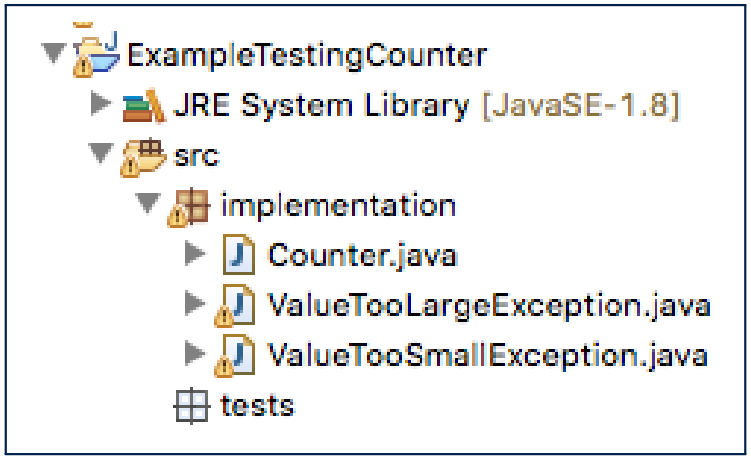
Why JUnit?

- *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.

How to Use JUnit: Packages

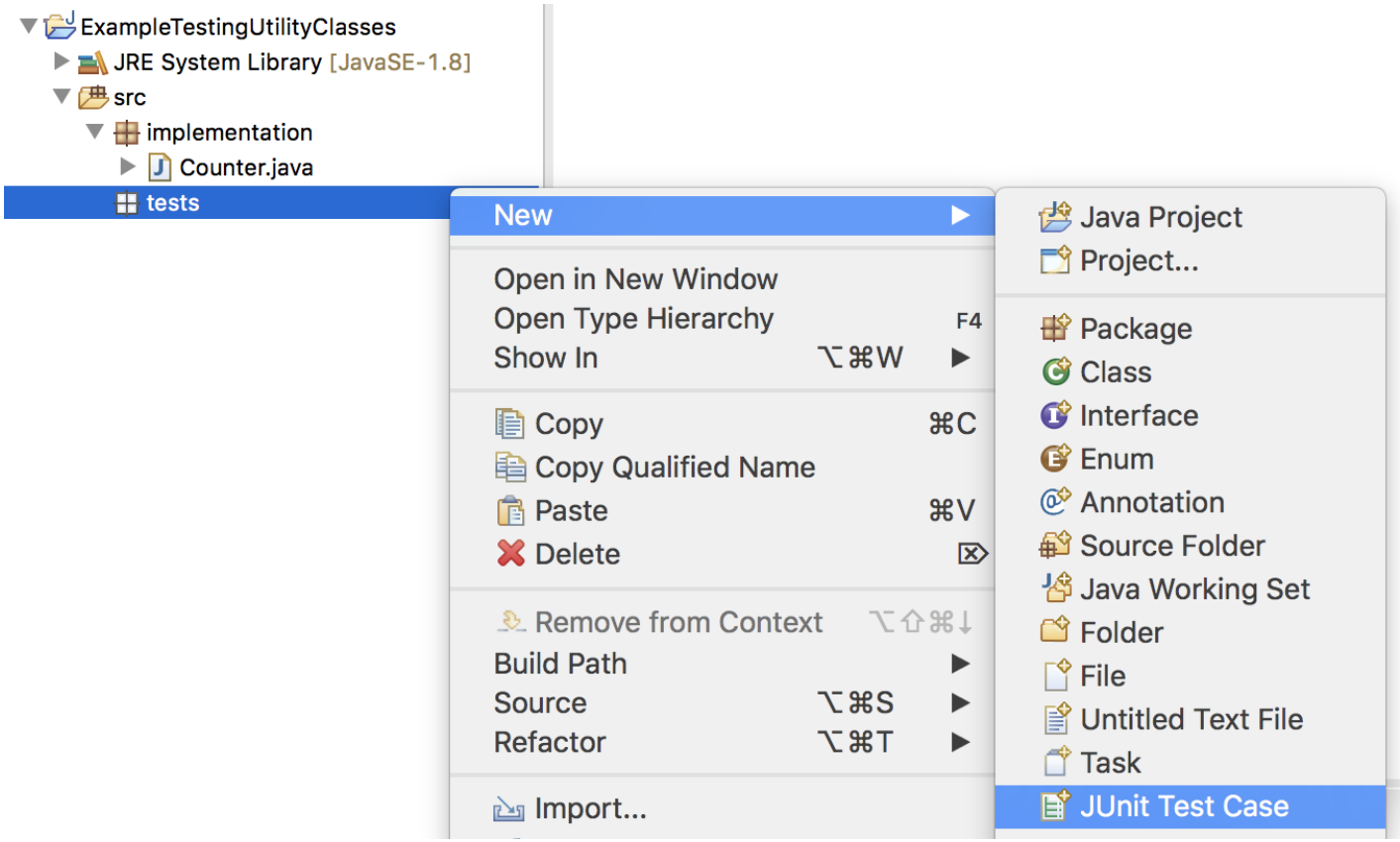
Step 1:

- 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 *JUnit Test Case* in `tests` package.

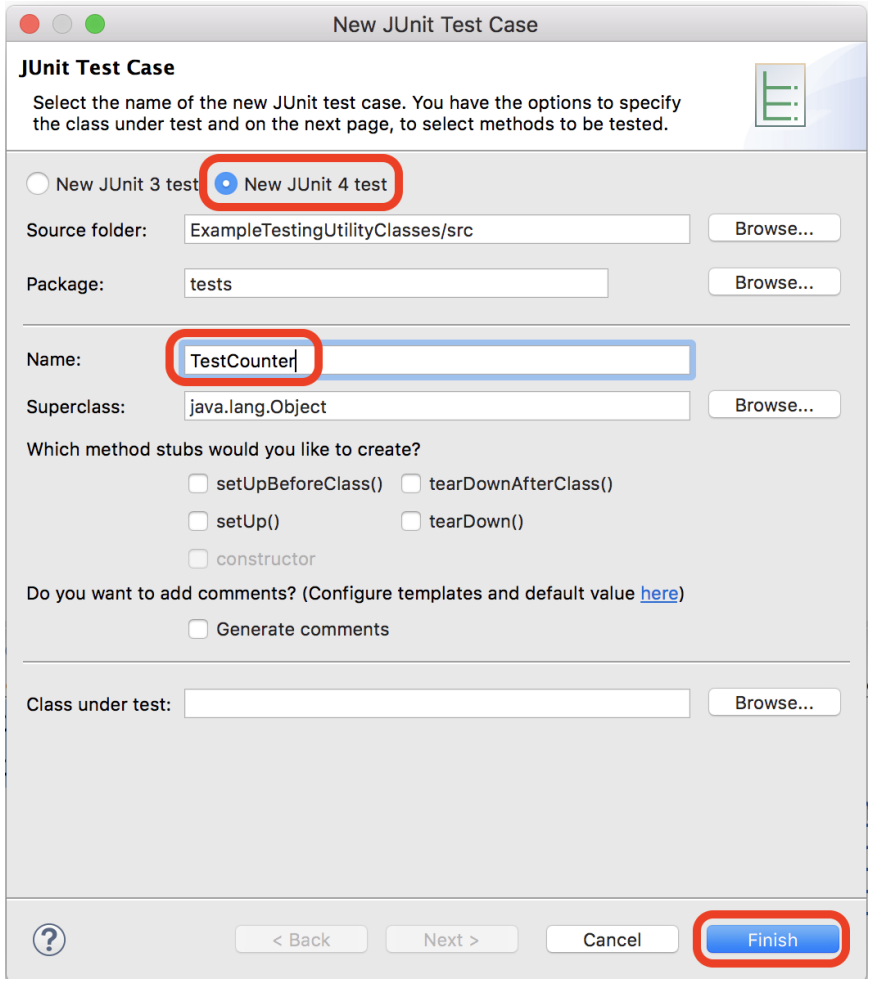


Create one JUnit Test Case to test one Java class only.

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

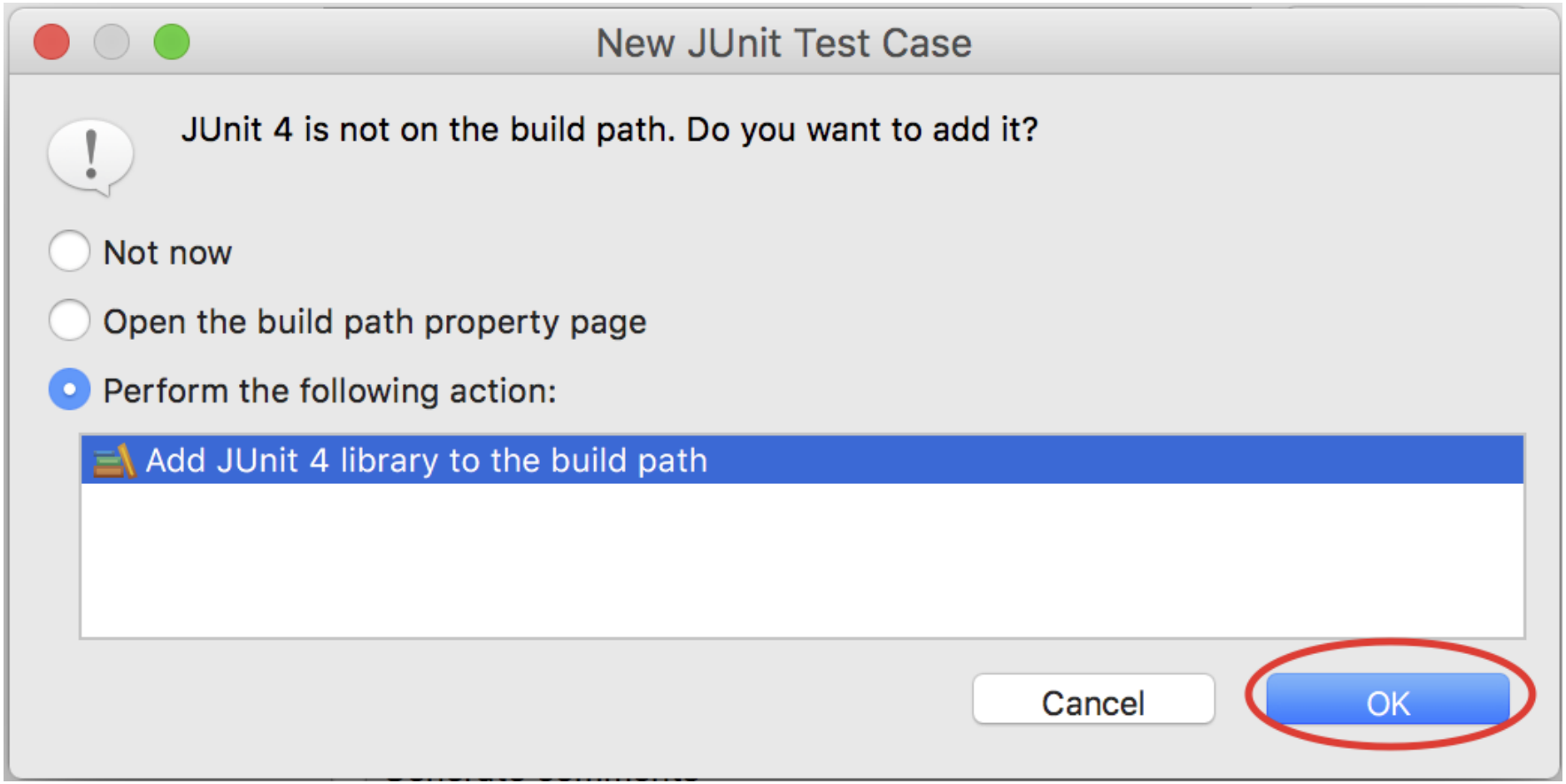
How to Use JUnit: New JUnit Test Case (2)

Step 3: Select the version of JUnit (JUnit 4); Enter the name of test case (`TestCounter`); Finish 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.



How to Use JUnit: Generated Test Case

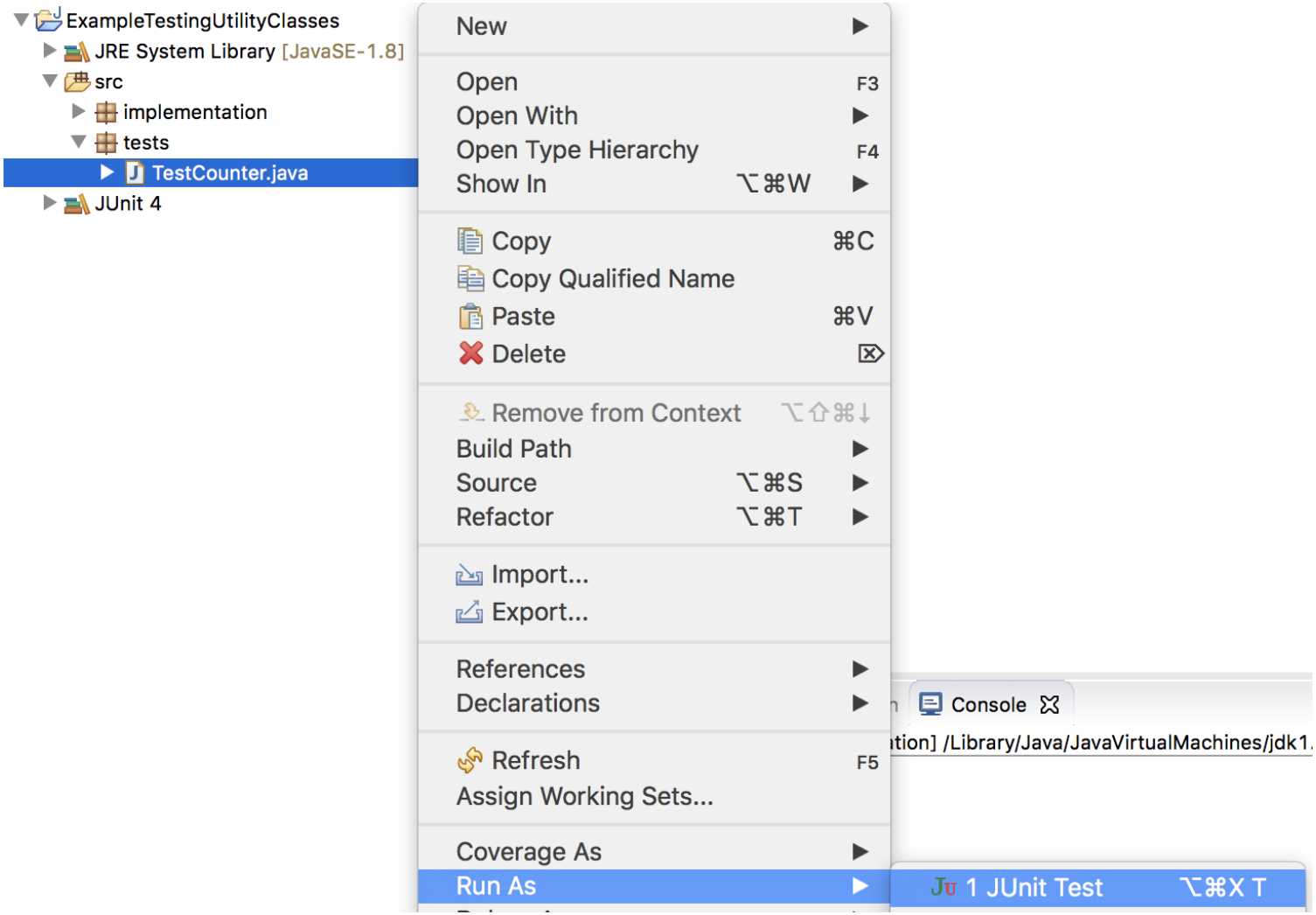
TestCounter.java

```
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".

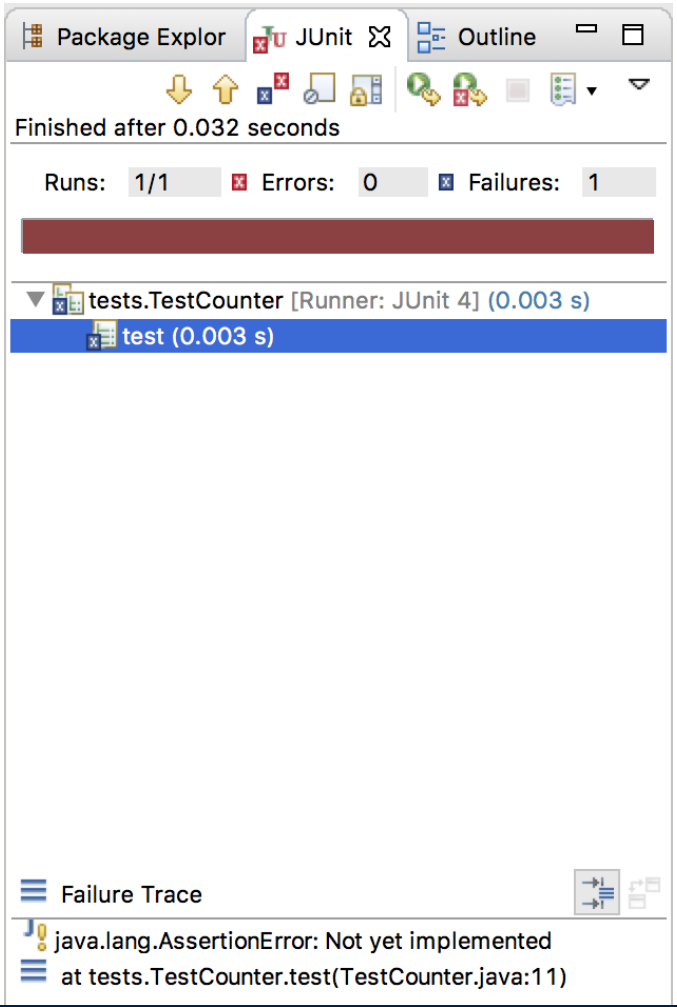
How to Use JUnit: Running Test Case

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")`.

How to Use JUnit: Revising Test Case

TestCounter.java

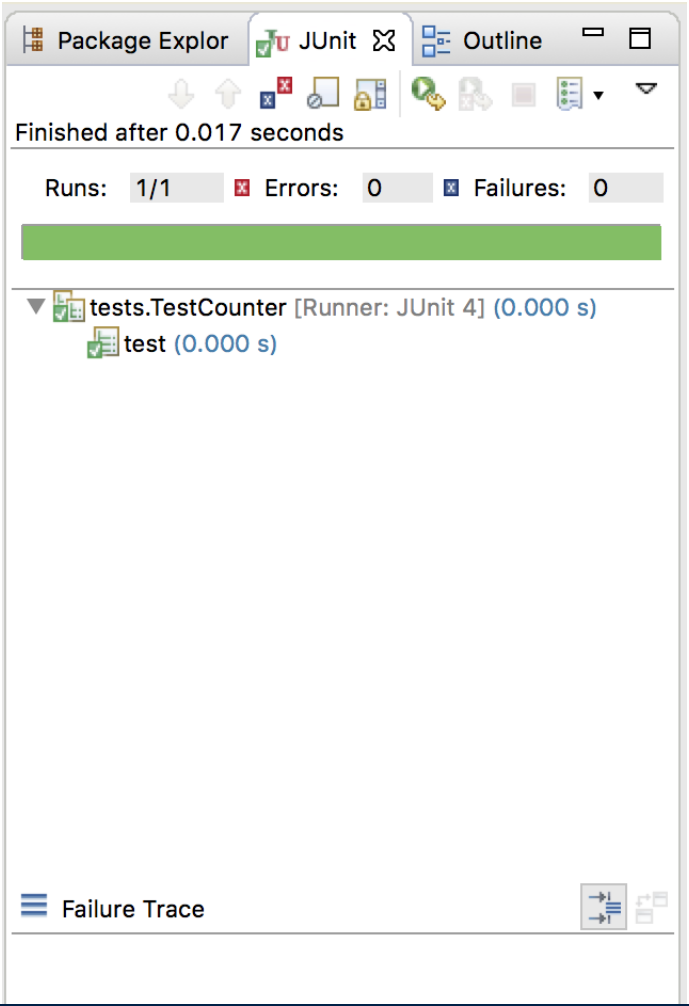
```
1 package tests;
2 import static org.junit.Assert.*;
3 import org.junit.Test;
4 public class TestCounter {
5     @Test
6     public void test() {
7 //      fail("Not yet implemented");
8     }
9 }
```

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`.



How to Use JUnit: Adding More Tests (1)

- 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(expected, actual)`
  - `void assertArrayEquals(expecteds, actuals)`
  - `void assertTrue(boolean condition)`
  - `void fail(String message)`

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.

How to Use JUnit: Adding More Tests (2.1)

```
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    } catch(ValueTooBigException e) {
9      /* Exception is not expected to be thrown. */
10     fail("ValueTooBigException is not expected."); } }
```

- **Lines 5 & 8**: We need a try-catch block because of **Line 6**.

  Method `increment` from class `Counter` may throw the `ValueTooBigException`.
- **Lines 4, 7 & 10** are all assertions:
  - **Lines 4 & 7** assert that `c.getValue()` returns the expected values.
  - **Line 10**: 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!*

How to Use JUnit: Adding More Tests (3.1)

```
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    catch(ValueTooSmallException e) {
9      /* Exception is expected to be thrown. */ } }
```

- **Lines 5 & 8**: We need a try-catch block because of **Line 6**.
  Method `decrement` from class `Counter` may throw the `ValueTooSmallException`.
- **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!
- 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!
```

- Again, *automation* is exactly rationale behind using JUnit!

How to Use JUnit: Adding More Tests (4.1)

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

- **Lines 4 – 8**:
  We use a try-catch block to express that a VTLE *is not* expected.
- **Lines 9 – 15**:
  We use a try-catch block to express that a VTLE *is* expected.

How to Use JUnit: Adding More Tests (4.2)

- 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!
```

How to Use JUnit: Adding More Tests (4.3)

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   } catch (ValueTooLargeException e) { }
12 }
```

No!

At **Line 9**, 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*.

How to Use JUnit: Adding More Tests (5)

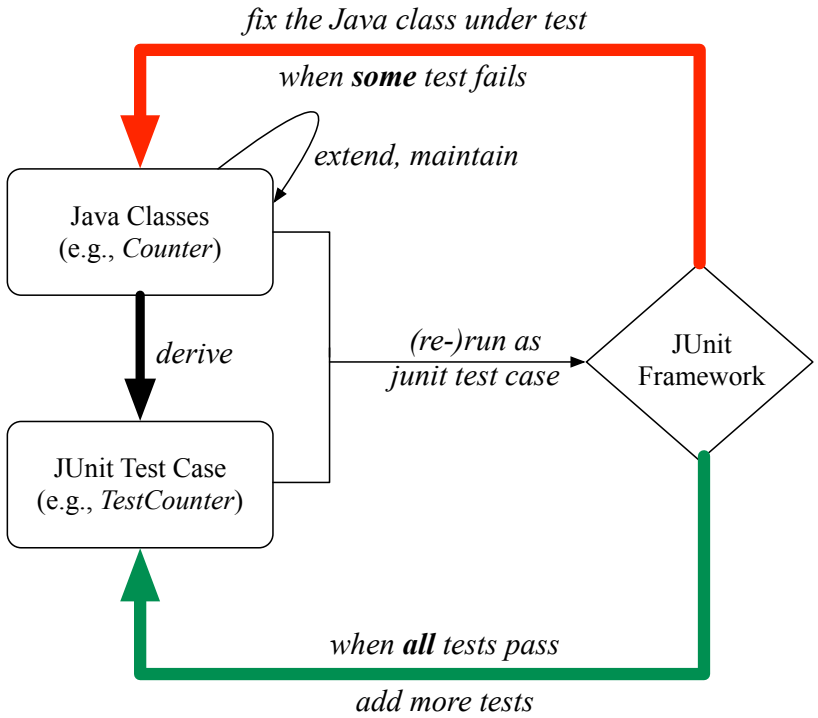
Loops can make it effective on generating test cases:

```
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   } catch(ValueTooLargeException e) {
16     fail("ValueTooLargeException is thrown unexpectedly");
17   } catch(ValueTooSmallException e) {
18     fail("ValueTooSmallException is thrown unexpectedly");
19   } }
```

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 error injection, and re-run all 8 tests. [What happens?]

Test-Driven Development (TDD)



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`

Index (1)

Index (2)

Index (3)

Advanced Topics on Classes and Objects

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

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 *another* new `Point` object
- The binary operator `==` may be applied to compare:
  - *Primitive* variables: their *contents* 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 *different* objects, even if their contents are *identical*.

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 *accessor method* 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);
}
```

    e.g., Say `p1` and `p2` are of type `Point`***V1*** without the `equals` method redefined, 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`.

Equality (2.2): Common Error

```
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.

Equality (3)

```
class PointV1 {
  double x; double y;
  PointV1(double x, double y) { this.x = x; this.y = y; }
}
```

```
1  PointV1 p1 = new PointV1(2, 3);
2  PointV1 p2 = new PointV1(2, 3);
3  System.out.println( p1 == p2 ); /* false */
4  System.out.println( p1.equals(p2) ); /* false */
```

- At **L4**, given that the `equals` method is not explicitly redefined/overridden in class `PointV1`, the default version inherited from class `Object` is called.
  Executing `p1.equals(p2)` boils down to (`p1 == p2`).
- If we wish to compare contents of two `PointV1` objects, need to explicitly redefine/override the `equals` method in that class.

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)

- How do we compare *contents* rather than addresses?
- Define the *accessor method* `equals`, e.g.,

```
class PointV2 {
  double x; double 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; } }
```

```
String s = "(2, 3)";
PointV2 p1 = new PointV2(2, 3); PointV2 p2 = new PointV2(2, 3);
System.out.println(p1.equals(p1));  /* true */
System.out.println(p1.equals(null));  /* false */
System.out.println(p1.equals(s));  /* false */
System.out.println(p1 == p2);  /* false */
System.out.println(p1.equals(p2));  /* true */
```

Equality (4.2)

- When making a method call `p.equals(o)`:
  - Variable `p` is declared of type `Point` ***V2***
  - Variable `o` can be declared of any type (e.g., `Point` ***V2***, `String`)
- We define `p` and `o` as ***equal*** if:
  - Either `p` and `o` refer to the same object;
  - Or:
    - `o` is not null.
    - `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 {
  boolean equals(Object obj) {
    if(this == null) { return false; }
```

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

Equality (4.3)

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

- `Object obj` at **L2** declares a parameter `obj` of type `Object`.
- `PointV2 other` at **L4** declares a variable `p` of type `PointV2`. We call such types declared at compile time as ***static type***.
- The list of *applicable attributes/methods* that we may call on 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 `Point`).
- If we are SURE that an object's "actual" type is different from its ***static type***, then we can *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 `Point`.
- Such cast allows more attributes/methods to be called upon `(Point) obj` at **L5**.

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 (6.1)

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

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

Q: At **L5**, will we get `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 *guard constraint* for the right operand `this.getClass() != obj.getClass()`.

Equality (6.2)

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

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

Q: At **L5**, if swapping the order of two operands of disjunction:

`this.getClass() != obj.getClass() || obj == null`

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

L10 & L11 call `equals` method defined in the `String` class.

When defining `equals` method for your own class, *reuse* `equals` methods defined in other classes wherever possible.

Equality (6.4)

Person collectors are equal if containing equal lists of persons.

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

Redefine/Override the `equals` method in `PersonCollector`.

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

Equality in JUnit (7.1)

- ***assertSame***(`obj1`, `obj2`)
  - Passes if `obj1` and `obj2` are references to the same object
  - ≈ assertTrue(`obj1 == obj2`)
  - ≈ assertFalse(`obj1 != obj2`)

```
PointV1 p1 = new PointV1(3, 4); PointV1 p2 = new PointV1(3, 4);
PointV1 p3 = p1;
assertSame(p1, p3); /* pass */ assertSame(p2, p3); /* fail */
```

- ***assertEquals***(`exp1`, `exp2`)
  - ≈ `exp1 == exp2` if `exp1` and `exp2` are **primitive** type

```
int i = 10; int j = 20; assertEquals(i, j); /* fail */
```

  - ≈ `exp1`.equals(`exp2`) if `exp1` and `exp2` are **reference** type

    Q: What if `equals` is not explicitly defined in *obj*1's declared type?

    A: ≈ assertSame(`obj1`, `obj2`)

```
PointV2 p4 = new PointV2(3, 4); PointV2 p5 = new PointV2(3, 4);
assertEquals(p4, p5); /* pass */
assertEquals(p1, p2); /* fail ∵ different PointV1 objects */
assertEquals(p4, p2); /* fail ∵ different types */
```

Equality in JUnit (7.2)

```
@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.x == p2.x && p2.y == p2.y);
}
@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, p4); */ /* 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 in JUnit (7.3)

```
@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.persons[0] == pc2.persons[0]);
  assertTrue(pc1.persons[0].equals(pc2.persons[0]));
  assertTrue(pc1.equals(pc2));

  pc1.addPerson(p3); pc2.addPerson(p4);
  assertTrue(pc1.persons[1] == pc2.persons[1]);
  assertTrue(pc1.persons[1].equals(pc2.persons[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.persons[2] == pc2.persons[2]);
  assertFalse(pc1.persons[2].equals(pc2.persons[2]));
  assertFalse(pc1.equals(pc2));
}
```

Why Ordering Between Objects? (1)

Each employee has their numerical id and salary.

e.g., $(alan,\ 2,\ 4500.34)$, $(mark,\ 3,\ 3450.67)$, $(tom,\ 1,\ 3450.67)$

- ***Problem***: To facilitate an annual review on their statuses, we want to arrange them so that ones with smaller id's come before ones with larger id's.s

  e.g., $\langle tom, alan, mark \rangle$

- Even better, arrange them so that ones with larger salaries come first; only compare id's for employees with equal salaries.

  e.g., $\langle alan, tom, mark \rangle$

- *Solution*:
  - Define ***ordering*** of `Employee` objects.
    [`Comparable` interface, `compareTo` method]
  - Use the library method `Arrays.sort`.

Why Ordering Between Objects? (2)

```
class Employee {
  int id; double salary;
  Employee(int id) { this.id = id; }
  void setSalary(double salary) { this.salary = salary; } }
```

```
1  @Test
2  public void testUncomparableEmployees() {
3    Employee alan = new Employee(2);
4    Employee mark = new Employee(3);
5    Employee tom = new Employee(1);
6    Employee[] es = {alan, mark, tom};
7    Arrays.sort(es);
8    Employee[] expected = {tom, alan, mark};
9    assertArrayEquals(expected, es); }
```

L8 triggers a ***java.lang.ClassCastException***:

Employee cannot be cast to java.lang.Comparable

∵ `Arrays.sort` expects an array whose element type defines a precise ***ordering*** of its instances/objects.

Defining Ordering Between Objects (1.1)

```
class CEmployee1  implements Comparable<CEmployee1> {
  ... /* attributes, constructor, mutator similar to Employee */
  @Override
  public int compareTo(CEmployee1 e) { return this.id - e.id; }
}
```

- Given two `CEmployee1` objects `ce1` and `ce2`:
  - `ce1.compareTo(ce2) > 0` [`ce1` "is greater than" `ce2`]
  - `ce1.compareTo(ce2) == 0` [`ce1` "is equal to" `ce2`]
  - `ce1.compareTo(ce2) < 0` [`ce1` "is smaller than" `ce2`]
- Say `ces` is an array of `CEmployee1` (`CEmployee1[] ces`), calling `Arrays.sort(ces)` re-arranges `ces`, so that:

$$\underbrace{\texttt{ces[0]}}_{\texttt{CEmployee1 } \textit{object}} \leq \underbrace{\texttt{ces[1]}}_{\texttt{CEmployee1 } \textit{object}} \leq \ldots \leq \underbrace{\texttt{ces[ces.length - 1]}}_{\texttt{CEmployee1 } \textit{object}}$$

Defining Ordering Between Objects (1.2)

```
@Test
public void testComparableEmployees_1() {
  /*
   * CEmployee1 implements the Comparable interface.
   * Method compareTo compares id's only.
   */
  CEmployee1 alan = new CEmployee1(2);
  CEmployee1 mark = new CEmployee1(3);
  CEmployee1 tom = new CEmployee1(1);
  alan.setSalary(4500.34);
  mark.setSalary(3450.67);
  tom.setSalary(3450.67);
  CEmployee1[] es = {alan, mark, tom};
  /* When comparing employees,
   * their salaries are irrelevant.
   */
  Arrays.sort(es);
  CEmployee1[] expected = {tom, alan, mark};
  assertArrayEquals(expected, es);
}
```

Defining Ordering Between Objects (2.1)

Let's now make the comparison more sophisticated:

- Employees with higher salaries come before those with lower salaries.
- When two employees have same salary, whoever with lower id comes first.

```
1  class CEmployee2  implements Comparable<CEmployee2> {
2    ... /* attributes, constructor, mutator similar to Employee */
3    @Override
4    public int compareTo(CEmployee2 other) {
5      int salaryDiff = Double.compare(this.salary, other.salary);
6      int idDiff = this.id - other.id;
7      if(salaryDiff != 0) { return -salaryDiff; }
8      else { return idDiff; } } }
```

- **L5**: `Double.compare(d1, d2)` returns
  - (`d1 < d2`), 0 (`d1 == d2`), or + (`d1 > d2`).
- **L7**: Why inverting the sign of `salaryDiff`?
  - *this*.*salary* > *other*.*salary* ⇒ `Double.compare`(*this*.*salary*, *other*.*salary*) > 0
  - But we should consider employee with *higher* salary as "smaller".
    ∴ We want that employee to come *before* the other one!

Defining Ordering Between Objects (2.2)

Alternatively, we can use extra `if` statements to express the logic more clearly.

```
1  class CEmployee2  implements Comparable<CEmployee2> {
2    ... /* attributes, constructor, mutator similar to Employee */
3    @Override
4    public int compareTo(CEmployee2 other) {
5      if(this.salary > other.salary) {
6        return -1;
7      }
8      else if (this.salary < other.salary) {
9        return 1;
10     }
11     else { /* equal salaries */
12       return this.id - other.id;
13     }
14 }
```

Defining Ordering Between Objects (2.3)

```
1  @Test
2  public void testComparableEmployees_2() {
3    /*
4     * CEmployee2 implements the Comparable interface.
5     * Method compareTo first compares salaries, then
6     * compares id's for employees with equal salaries.
7     */
8    CEmployee2 alan = new CEmployee2(2);
9    CEmployee2 mark = new CEmployee2(3);
10   CEmployee2 tom = new CEmployee2(1);
11   alan.setSalary(4500.34);
12   mark.setSalary(3450.67);
13   tom.setSalary(3450.67);
14   CEmployee2[] es = {alan, mark, tom};
15   Arrays.sort(es);
16   CEmployee2[] expected = {alan, tom, mark};
17   assertArrayEquals(expected, es);
18 }
```

Defining Ordering Between Objects (3)

When you have your class `C` implement the interface `Comparable<C>`, you should design the `compareTo` method, such that given objects `c1`, `c2`, `c3` of type `C`:

- *Asymmetric*:

$$\neg(c1.compareTo(c2) < 0 \wedge c2.compareTo(c1) < 0)$$
$$\neg(c1.compareTo(c2) > 0 \wedge c2.compareTo(c1) > 0)$$

$\therefore$ We don't have $c1 < c2$ and $c2 < c1$ at the same time!

- *Transitive*:

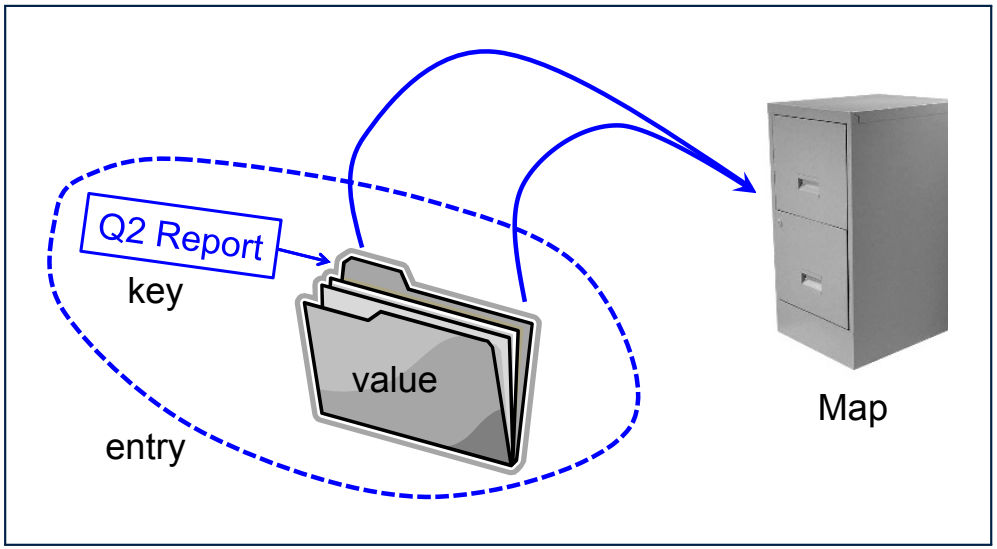
$$c1.compareTo(c2) < 0 \wedge c2.compareTo(c3) < 0 \Rightarrow c1.compareTo(c3) < 0$$
$$c1.compareTo(c2) > 0 \wedge c2.compareTo(c3) > 0 \Rightarrow c1.compareTo(c3) > 0$$

$\therefore$ We have $c1 < c2 \wedge c2 < c3 \Rightarrow c1 < c3$

Q. How would you define the `compareTo` method for the `Player` class of a rock-paper-scissor game? [**Hint**: Transitivity]

Hashing: What is a Map?

- A *map* (a.k.a. table or dictionary) stores a collection of *entries*.



Entry	
(Search) Key	Value
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

- Each *entry* is a pair: a *value* and its *(search) key*.
- Each *search key*:
  - *Uniquely* identifies an object in the map
  - Should be used to *efficiently* retrieve the associated value
- Search keys must be *unique* (i.e., do not contain duplicates).

Hashing: Arrays are Maps

- Each array *entry* is a pair: an object and its *numerical* index.
  e.g., say `String[] a = {"A", "B", "C"}`, how many entries?
  3 entries: `(0, "A")`, `(1, "B")`, `(2, "C")`
- *Search keys* are the set of numerical index values.
- The set of index values are *unique* [e.g., 0 .. (*a.length* – 1)]
- Given a *valid* index value *i*, we can
  - *Uniquely* determines where the object is [$(i+1)^{th}$ item]
  - *Efficiently* retrieves that object [`a[i]` ≈ fast memory access]
- Maps in general may have *non-numerical* key values:
  - Student ID [student record]
  - Social Security Number [resident record]
  - Passport Number [citizen record]
  - Residential Address [household record]
  - Media Access Control (MAC) Address [PC/Laptop record]
  - Web URL [web page]

  . . .

Hashing: Naive Implementation of Map

- **Problem**: Support the construction of this simple map:

Entry	
(Search) Key	Value
1	D
25	C
3	F
14	Z
6	A
39	C
7	Q

  Let's just assume that the maximum map capacity is 100.

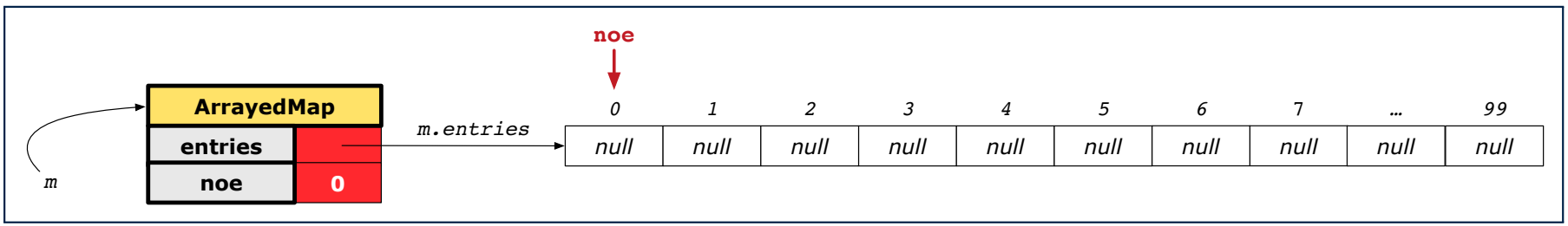
- **Naive Solution**:

  Let's understand the expected runtime structures before seeing the Java code!

Hashing: Naive Implementation of Map (0)

After executing `ArrayedMap m = new ArrayedMap()`:

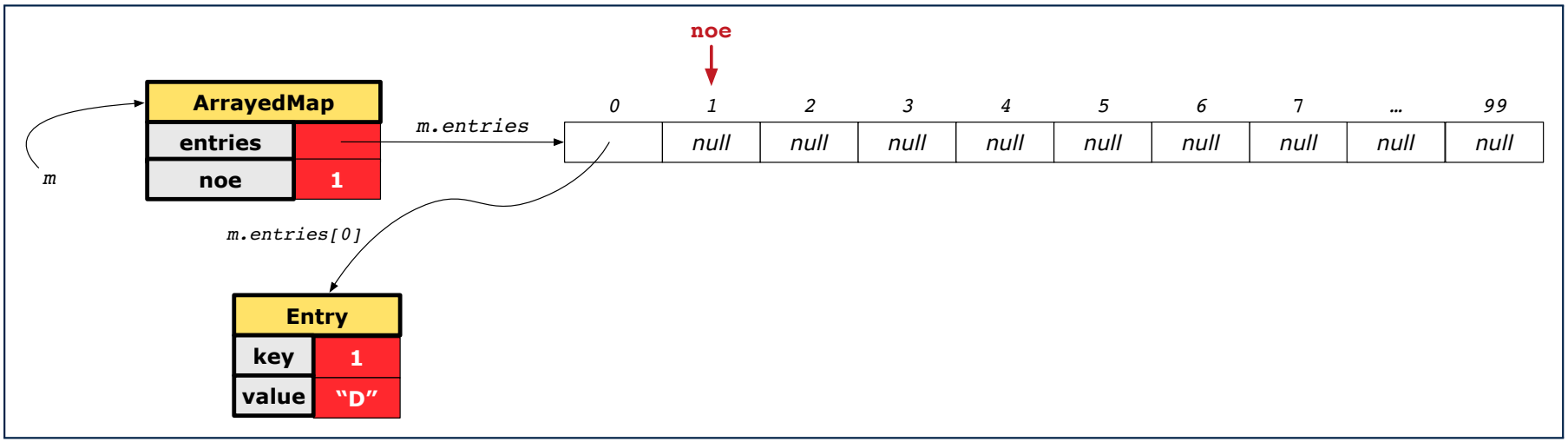
- Attribute `m.entries` initialized as an array of 100 `null` slots.
- Attribute `m.noe` is 0, meaning:
  - Current number of entries stored in the map is 0.
  - Index for storing the next new entry is 0.



Hashing: Naive Implementation of Map (1)

After executing `m.put(new Entry(1, "D"))`:

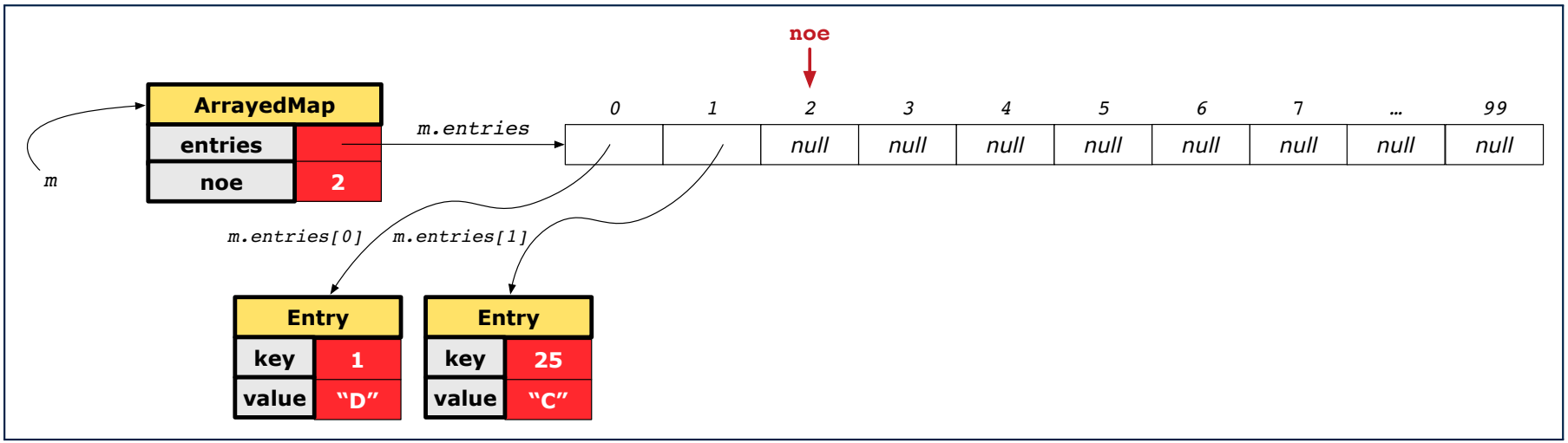
- Attribute `m.entries` has 99 `null` slots.
- Attribute `m.noe` is 1, meaning:
  - Current number of entries stored in the map is 1.
  - Index for storing the next new entry is 1.



Hashing: Naive Implementation of Map (2)

After executing `m.put(new Entry(25, "C"))`:

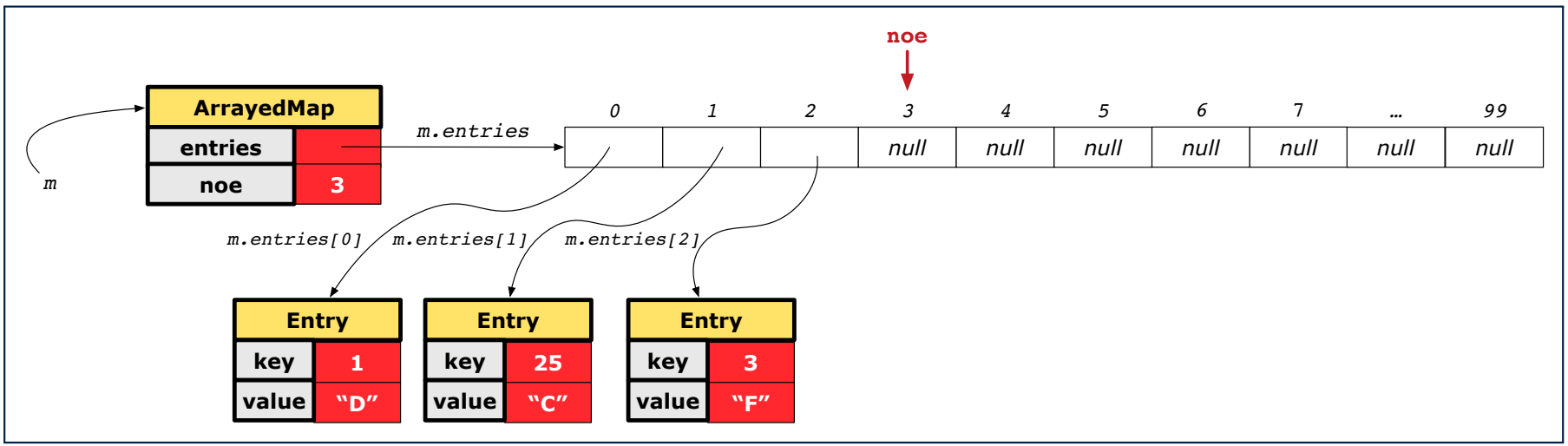
- Attribute `m.entries` has 98 `null` slots.
- Attribute `m.noe` is 2, meaning:
  - Current number of entries stored in the map is 2.
  - Index for storing the next new entry is 2.



Hashing: Naive Implementation of Map (3)

After executing `m.put(new Entry(3, "F"))`:

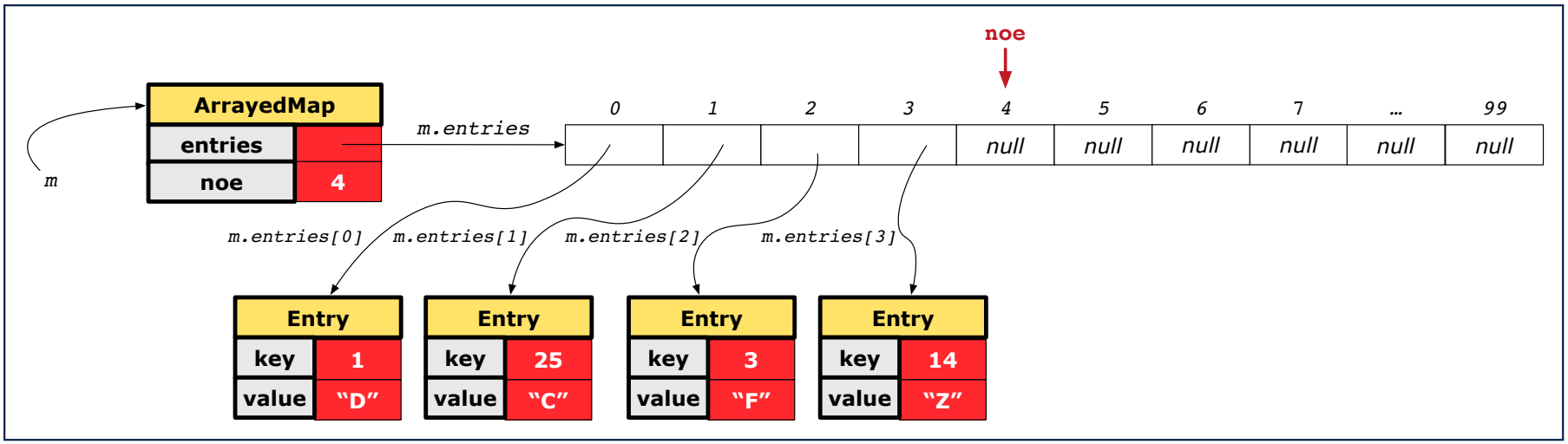
- Attribute `m.entries` has 97 `null` slots.
- Attribute `m.noe` is 3, meaning:
  - Current number of entries stored in the map is 3.
  - Index for storing the next new entry is 3.



Hashing: Naive Implementation of Map (4)

After executing `m.put(new Entry(14, "Z"))`:

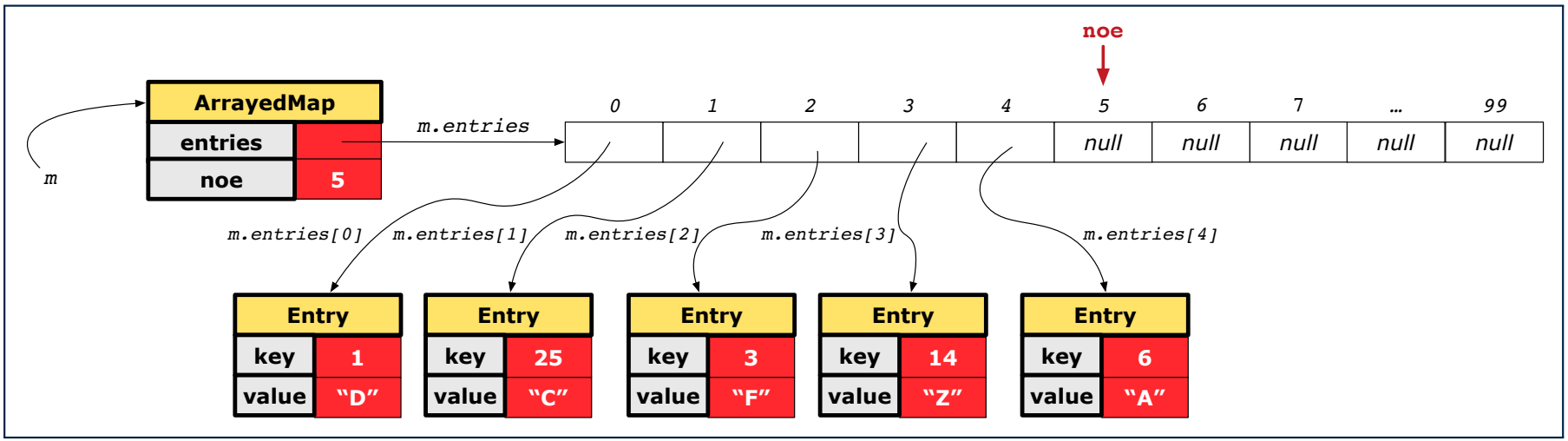
- Attribute `m.entries` has 96 `null` slots.
- Attribute `m.noe` is 4, meaning:
  - Current number of entries stored in the map is 4.
  - Index for storing the next new entry is 4.



Hashing: Naive Implementation of Map (5)

After executing `m.put(new Entry(6, "A"))`:

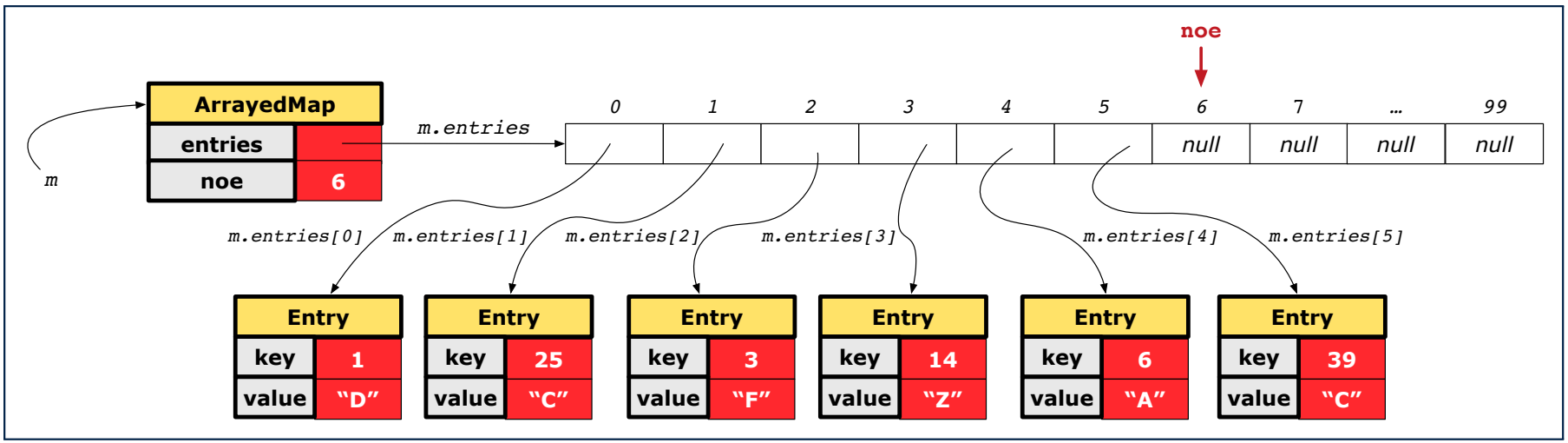
- Attribute `m.entries` has 95 `null` slots.
- Attribute `m.noe` is 5, meaning:
  - Current number of entries stored in the map is 5.
  - Index for storing the next new entry is 5.



Hashing: Naive Implementation of Map (6)

After executing `m.put(new Entry(39, "C"))`:

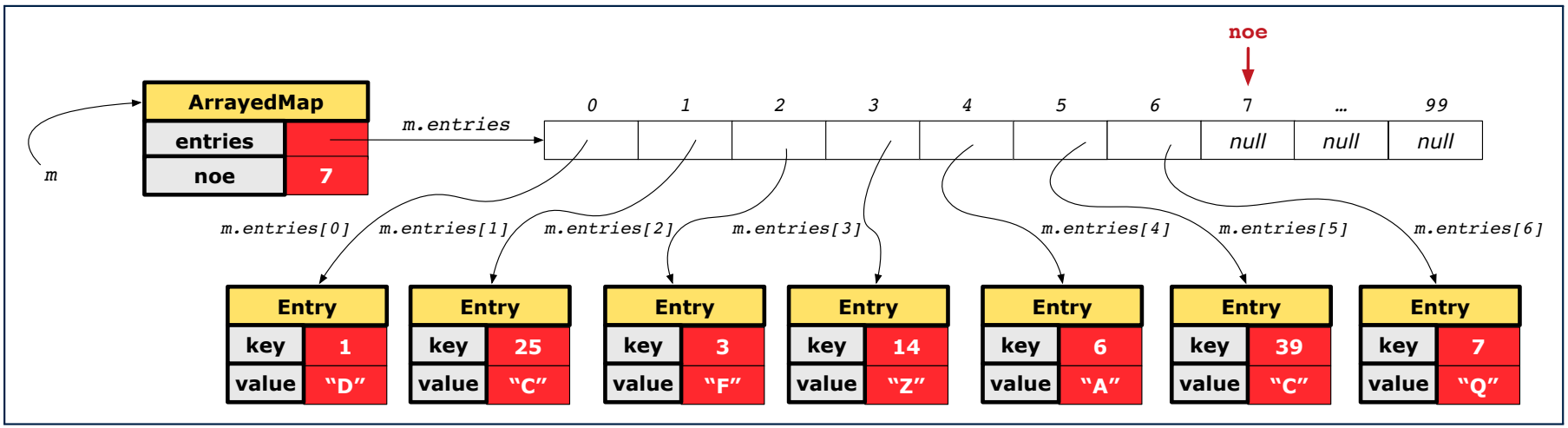
- Attribute `m.entries` has 94 `null` slots.
- Attribute `m.noe` is 6, meaning:
  - Current number of entries stored in the map is 6.
  - Index for storing the next new entry is 6.



Hashing: Naive Implementation of Map (7)

After executing `m.put(new Entry(7, "Q"))`:

- Attribute `m.entries` has 93 `null` slots.
- Attribute `m.noe` is 7, meaning:
  - Current number of entries stored in the map is 7.
  - Index for storing the next new entry is 7.



Hashing: Naive Implementation of Map (8.1)

```
public class Entry {
  private int key;
  private String value;

  public Entry(int key, String value) {
    this.key = key;
    this.value = value;
  }
  /* Getters and Setters for key and value */
}
```

Hashing: Naive Implementation of Map (8.2)

```
public class ArrayedMap {
  private final int MAX_CAPCAITY = 100;
  private Entry[] entries;
  private int noe; /* number of entries */
  public ArrayedMap() {
    entries = new Entry[MAX_CAPCAITY];
    noe = 0;
  }
  public int size() {
    return noe;
  }
  public void put(int key, String value) {
    Entry e = new Entry(key, value);
    entries[noe] = e;
    noe ++;
  }
```

Required Reading: Point and PointCollector

Hashing: Naive Implementation of Map (8.3)

```
@Test
public void testArrayedMap() {
  ArrayedMap m = new ArrayedMap();
  assertTrue(m.size() == 0);
  m.put(1, "D");
  m.put(25, "C");
  m.put(3, "F");
  m.put(14, "Z");
  m.put(6, "A");
  m.put(39, "C");
  m.put(7, "Q");
  assertTrue(m.size() == 7);
  /* inquiries of existing key */
  assertTrue(m.get(1).equals("D"));
  assertTrue(m.get(7).equals("Q"));
  /* inquiry of non-existing key */
  assertTrue(m.get(31) == null);
}
```

Hashing: Naive Implementation of Map (8.4)

```
public class ArrayedMap {
  private final int MAX_CAPCAITY = 100;
  public String get (int key) {
    for(int i = 0; i < noe; i ++) {
      Entry e = entries[i];
      int k = e.getKey();
      if(k == key) { return e.getValue(); }
    }
    return null;
  }
```

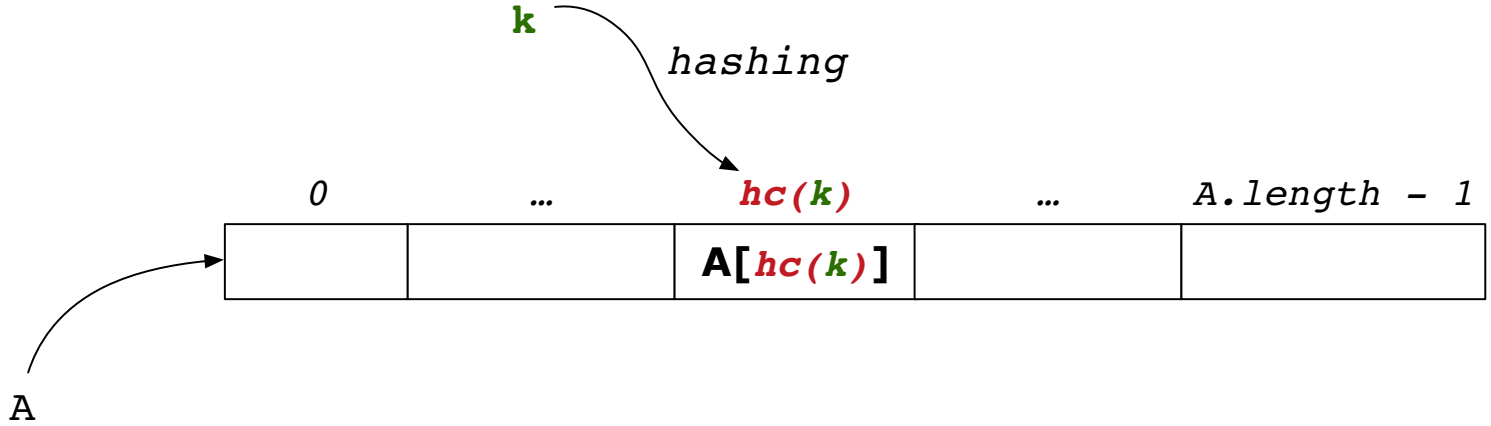
Say `entries` is: {(1, D), (25, C), (3, F), (14, Z), (6, A), (39, C), (7, Q), null, . . . }

- How efficient is `m.get(1)`? [1 iteration]
- How efficient is `m.get(7)`? [7 iterations]
- If `m` is full, worst case of `m.get(k)`? [100 iterations]
- If `m` with 10^6 entries, worst case of `m.get(k)`? [10^6 iterations]

⇒ `get`'s worst-case performance is *linear* on size of `m.entries`!

A much *faster* (and ***correct***) solution is possible!

Hashing: Hash Table (1)

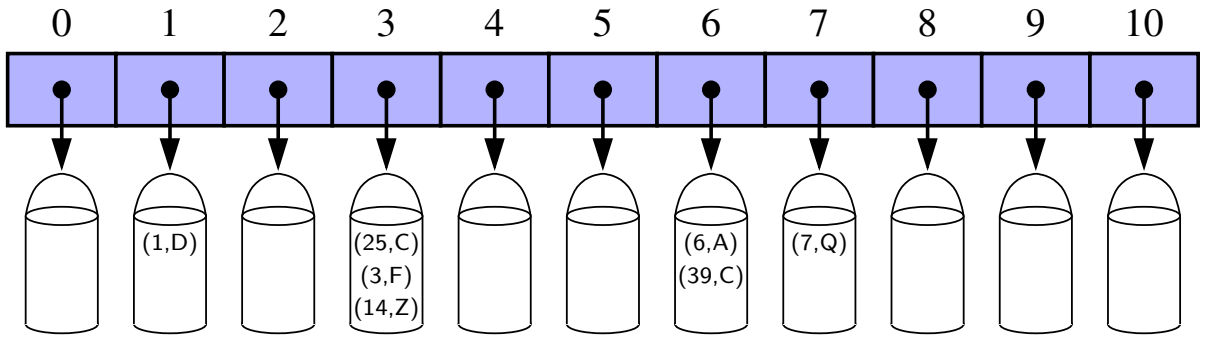


- Given a (numerical or non-numerical) search key ***k***:
  - Apply a function ***hc*** so that ***hc(k)*** returns an integer.
    - We call ***hc(k)*** the *hash code* of key ***k***.
    - Value of ***hc(k)*** denotes a ***valid index*** of some array `A`.
  - Rather than searching through array `A`, go directly to `A[` ***hc***(***k***) `]` to get the associated value.
- Both computations are fast:
  - Converting ***k*** to ***hc(k)***
  - Indexing into `A[` ***hc***(***k***) `]`

Hashing: Hash Table as a Bucket Array (2.1)

For illustration, assume `A.length` is 11 and $hc(k) = k\%11$.

$hc(k) = k\%11$	(SEARCH) KEY	VALUE
1	1	D
3	25	C
3	3	F
3	14	Z
6	6	A
6	39	C
7	7	Q

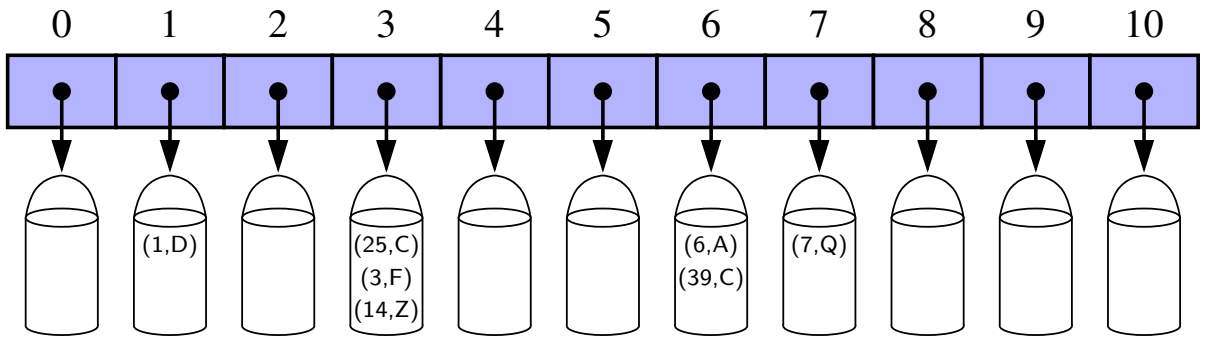


- **Collision**: unequal keys have same hash code (e.g., 25, 3, 14)
  ⇒ Unavoidable as number of entries ↑, but a *good* hash function should have sizes of the buckets uniformly distributed.

Hashing: Hash Table as a Bucket Array (2.2)

For illustration, assume `A.length` is 10 and $hc(k) = k\%11$.

$hc(k) = k\%11$	(Search) Key	Value
1	1	D
3	25	C
3	3	F
3	14	Z
6	6	A
6	39	C
7	7	Q



- **Collision**: unequal keys have same hash code (e.g., 25, 3, 14)
  ⇒ When there are *multiple entries* in the *same bucket*, we distinguish between them using their ***unequal*** keys.

Hashing: Contract of Hash Function

- Principle of defining a hash function ***hc***:

$$k1.equals(k2) \Rightarrow hc(k1) == hc(k2)$$

  Equal keys always have the same hash code.
- Equivalently, according to contrapositive:

$$hc(k1) \neq hc(k2) \Rightarrow \neg k1.equals(k2)$$

  Different hash codes must be generated from unequal keys.
- What if $\neg k1.equals(k2)$?
  - $hc(k1) == hc(k2)$ [collision e.g., 25 and 3]
  - $hc(k1) \neq hc(k2)$ [no collision e.g., 25 and 1]
- What if $hc(k1) == hc(k2)$?
  - $\neg k1.equals(k2)$ [collision e.g., 25 and 3]
  - $k1.equals(k2)$ [sound hash function]

inconsistent `hashCode` and `equals`

Hashing: Defining Hash Function in Java (1)

The `Object` class (common super class of all classes) has the method for redefining the hash function for your own class:

```
1  public class IntegerKey {
2    private int k;
3    public IntegerKey(int k) { this.k = k; }
4    @Override
5    public int hashCode() { return k % 11; }
6    @Override
7    public boolean equals(Object obj) {
8      if(this == obj) { return true; }
9      if(obj == null) { return false; }
10     if(this.getClass() != obj.getClass()) { return false; }
11     IntegerKey other = (IntegerKey) obj;
12     return this.k == other.k;
13   } }
```

Q: Can we replace **L12** by `return this.hashCode() == other.hashCode()`?

A: *No* ∵ When collision happens, keys with same hash code (i.e., in the same bucket) cannot be distinguished.

Hashing: Defining Hash Function in Java (2)

```
@Test
public void testCustomizedHashFunction() {
  IntegerKey ik1 = new IntegerKey(1);
  /* 1 % 11 == 1 */
  assertTrue(ik1.hashCode() == 1);

  IntegerKey ik39_1 = new IntegerKey(39); /* 39 % 11 == 6 */
  IntegerKey ik39_2 = new IntegerKey(39);
  IntegerKey ik6 = new IntegerKey(6); /* 6 % 11 == 6 */

  assertTrue(ik39_1.hashCode() == 6);
  assertTrue(ik39_2.hashCode() == 6);
  assertTrue(ik6.hashCode() == 6);

  assertTrue(ik39_1.hashCode() == ik39_2.hashCode());
  assertTrue(ik39_1.equals(ik39_2));

  assertTrue(ik39_1.hashCode() == ik6.hashCode());
  assertFalse(ik39_1.equals(ik6));
}
```

Hashing: Using Hash Table in Java

```
@Test
public void testHashTable() {
  Hashtable<IntegerKey, String> table = new Hashtable<>();
  IntegerKey k1 = new IntegerKey(39);
  IntegerKey k2 = new IntegerKey(39);
  assertTrue(k1.equals(k2));
  assertTrue(k1.hashCode() == k2.hashCode());
  table.put(k1, "D");
  assertTrue(table.get(k2).equals("D"));
}
```

Hashing: Defining Hash Function in Java (3)

- When you are given instructions as to how the `hashCode` method of a class should be defined, override it manually.
- Otherwise, use Eclipse to generate the `equals` and `hashCode` methods for you.
  - Right click on the class.
  - Select `Source`.
  - Select `Generate hashCode() and equals()`.
  - Select the relevant attributes that will be used to compute the hash value.

Hashing:
Defining Hash Function in Java (4.1.1)

Caveat: Always make sure that the `hashCode` and `equals` are redefined/overridden to work together consistently.

e.g., Consider an alternative version of the `IntegerKey` class:

```
public class IntegerKey {
  private int k;
  public IntegerKey(int k) { this.k = k; }
  /* hashCode() inherited from Object NOT overridden.  */
  @Override
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    IntegerKey other = (IntegerKey) obj;
    return this.k == other.k;
  } }
```

Hashing: Defining Hash Function in Java (4.1.2)

```
public class IntegerKey {
  private int k;
  public IntegerKey(int k) { this.k = k; }
  /* hashCode() inherited from Object NOT overridden.  */
  @Override
  public boolean equals(Object obj) {
    if(this == obj) { return true; }
    if(obj == null) { return false; }
    if(this.getClass() != obj.getClass()) { return false; }
    IntegerKey other = (IntegerKey) obj;
    return this.k == other.k;
  } }
```

- ***Problem***?
  - Default implementation of `hashCode()` from the `Object` class:
    Objects with *distinct* addresses have *distinct* hash code values.
  - Violation of the Contract of `hashCode()`:
    $hc(k1) \neq hc(k2) \Rightarrow \neg k1.equals(k2)$
- What about `equal` objects with different addresses?

Hashing: Defining Hash Function in Java (4.2)

```
1  @Test
2  public void testDefaultHashFunction() {
3    IntegerKey ik39_1 = new IntegerKey(39);
4    IntegerKey ik39_2 = new IntegerKey(39);
5    assertTrue(ik39_1.equals(ik39_2));
6    assertTrue(ik39_1.hashCode() != ik39_2.hashCode()); }
7  @Test
8  public void testHashTable() {
9    Hashtable<IntegerKey, String> table = new Hashtable<>();
10   IntegerKey k1 = new IntegerKey(39);
11   IntegerKey k2 = new IntegerKey(39);
12   assertTrue(k1.equals(k2));
13   assertTrue(k1.hashCode() != k2.hashCode());
14   table.put(k1, "D");
15   assertTrue(table.get(k2) == null); }
```

L3, 4, 10, 11: Default version of `hashCode`, inherited from `Object`, returns a *distinct* integer for every new object, *despite its contents*. [***Fix***: Override `hashCode` of your classes!]

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 for the method call.
  - Instead, argument variable **arg** is passed ***indirectly***: a *copy* of the value stored in **arg** is made and passed for 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:

```
class Point {
  int x;
  int y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  void moveVertically(int y){
    this.y += y;
  }
  void moveHorizontally(int x){
    this.x += x;
  }
}
```

Call by Value (2.2.1)

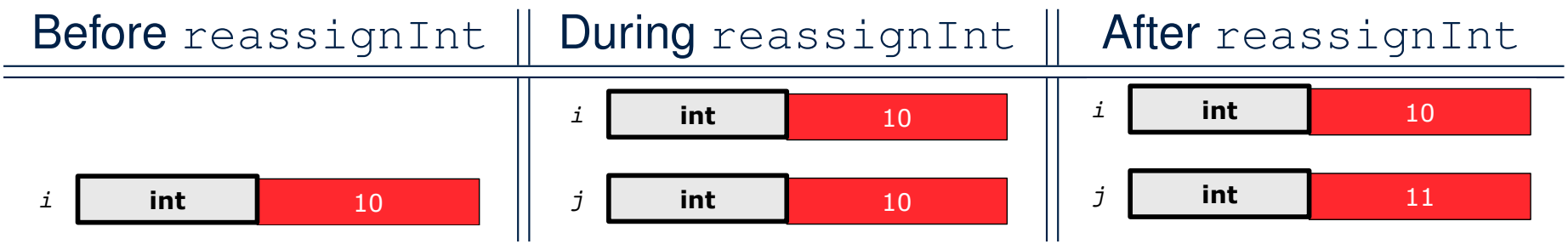
```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
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)



Call by Value (2.3.1)

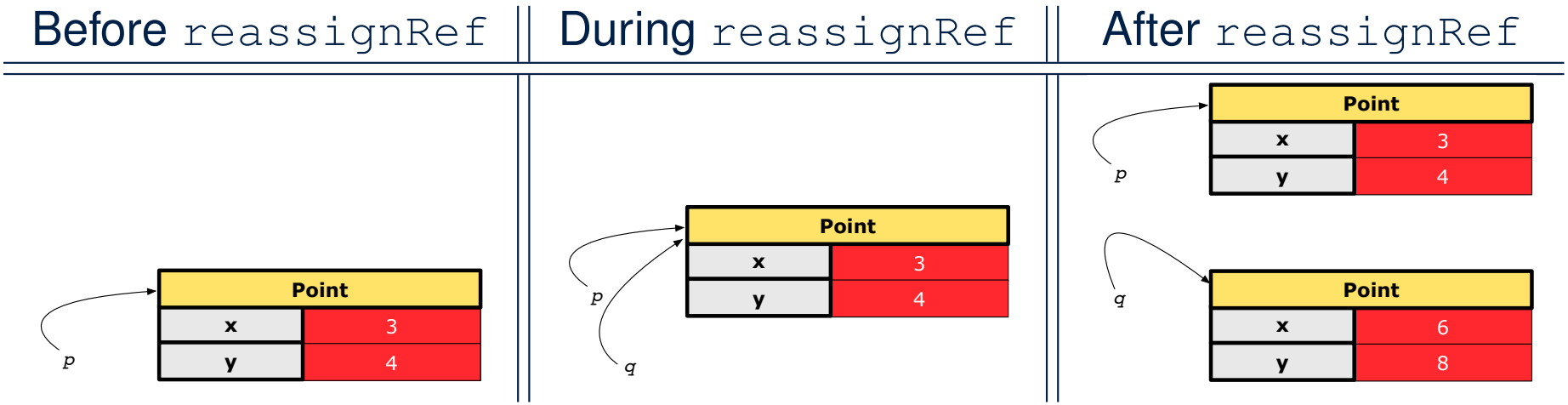
```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByRef_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.x==3 && p.y==4);
9  }
```

- ***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)



Call by Value (2.4.1)

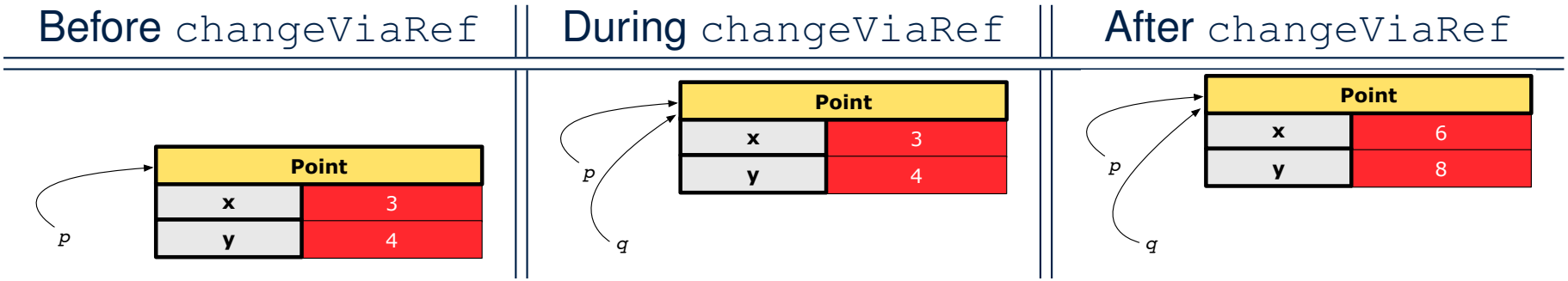
```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
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.x==6 && p.y==8);
9  }
```

- ***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.]

  ⇒ Calls to `q.moveHorizontally` and `q.moveVertically` are effective on both `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` have been modified via `q`.

Call by Value (2.4.2)



Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Asymptotic Analysis of Algorithms

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Algorithm and Data Structure

- A *data structure* is:
  - A systematic way to store and organize data in order to facilitate *access* and *modifications*
  - Never suitable for all purposes: it is important to know its *strengths* and *limitations*
- A *well-specified* *computational problem* precisely describes the desired *input/output relationship*.
  - **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - **Output:** A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
  - An *instance* of the problem: $\langle 3, 1, 2, 5, 4 \rangle$
- An *algorithm* is:
  - A solution to a well-specified *computational problem*
  - A *sequence of computational steps* that takes value(s) as *input* and produces value(s) as *output*
- Steps in an *algorithm* manipulate well-chosen *data structure(s)*.

Measuring "Goodness" of an Algorithm

1. *Correctness*:
   - Does the algorithm produce the expected output?
   - Use JUnit to ensure this.
2. Efficiency:
   - *Time Complexity*: processor time required to complete
   - *Space Complexity*: memory space required to store data

Correctness is always the priority.

How about efficiency? Is time or space more of a concern?

Measuring Efficiency of an Algorithm

- *Time* is more of a concern than is *storage*.
- Solutions that are meant to be run on a computer should run *as fast as possible*.
- Particularly, we are interested in how *running time* depends on two *input factors*:
  1. size
     e.g., sorting an array of 10 elements vs. 1m elements
  2. structure
     e.g., sorting an already-sorted array vs. a hardly-sorted array
- *How do you determine the running time of an algorithm?*
  1. Measure time via *experiments*
  2. Characterize time as a *mathematical function* of the input size

Measure Running Time via Experiments

- Once the algorithm is implemented in Java:
  - Execute the program on *test inputs* of various *sizes* and *structures*.
  - For each test, record the *elapsed time* of the execution.

```
long startTime = System.currentTimeMillis();
/* run the algorithm */
long endTime = System.currenctTimeMillis();
long elapsed = endTime - startTime;
```

  - *Visualize* the result of each test.
- To make *sound statistical claims* about the algorithm's *running time*, the set of input tests must be "reasonably" *complete*.

Example Experiment

- *Computational Problem*:
  - **Input**: A character *c* and an integer *n*
  - **Output**: A string consisting of *n* repetitions of character *c*
    e.g., Given input `'*'` and `15`, output `***************`.
- *Algorithm 1* using *String* Concatenations:

```
public static String repeat1(char c, int n) {
  String answer = "";
  for (int i = 0; i < n; i ++) { answer += c; }
  return answer; }
```

- *Algorithm 2* using *StringBuilder* `append`'s:

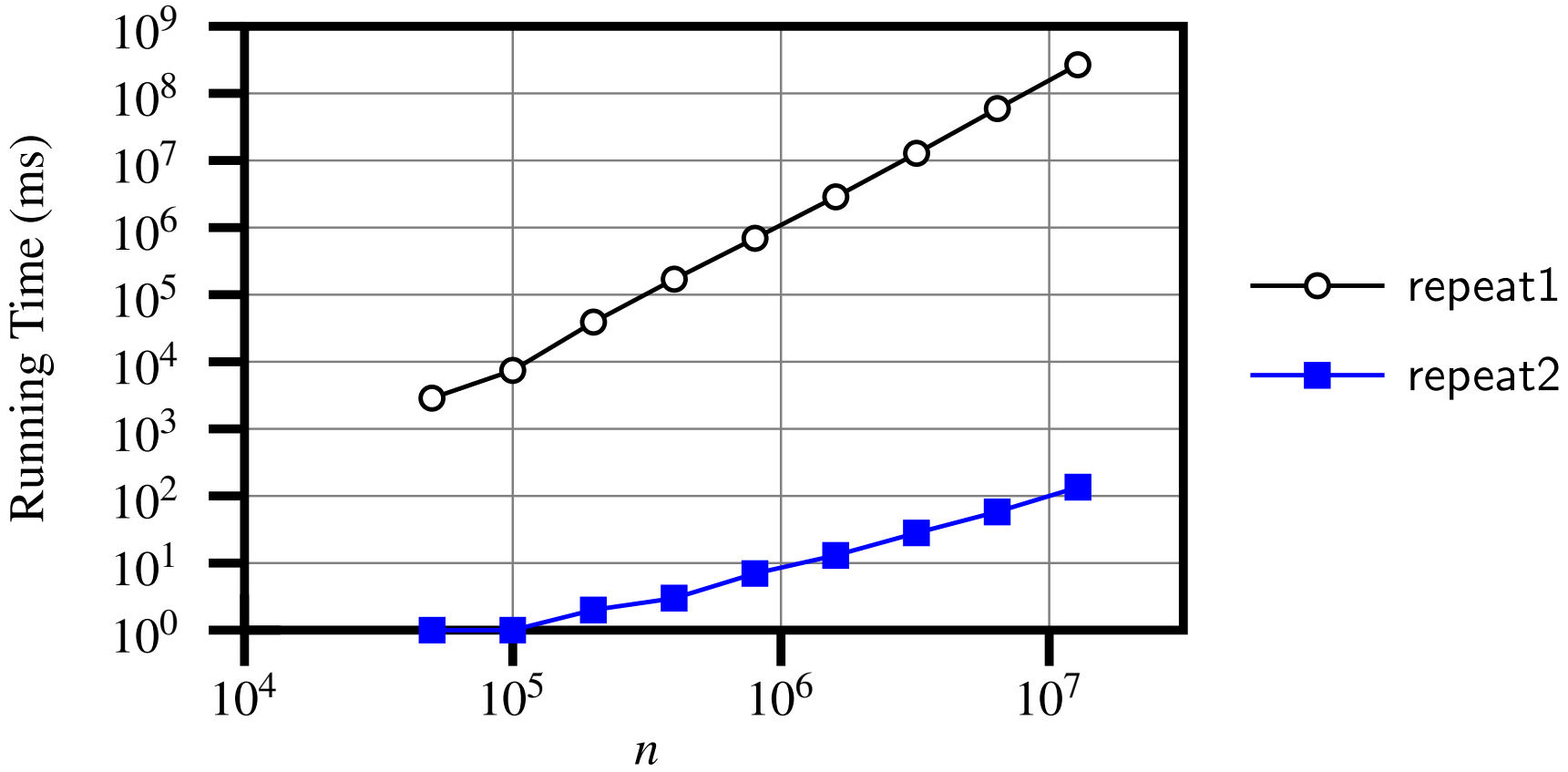
```
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int i = 0; i < n; i ++) { sb.append(c); }
  return sb.toString(); }
```

Example Experiment: Detailed Statistics

n	`repeat1` (in ms)	`repeat2` (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,847,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421 (≈ 3 days)	135

- As *input size* is doubled, *rates of increase* for both algorithms are *linear*:
  - *Running time* of `repeat1` increases by ≈ 5 times.
  - *Running time* of `repeat2` increases by ≈ 2 times.

Example Experiment: Visualization



Experimental Analysis: Challenges

1. An algorithm must be *fully implemented* (i.e., translated into valid Java syntax) in order study its runtime behaviour *experimentally*.
   - What if our purpose is to *choose among alternative* data structures or algorithms to implement?
   - Can there be a *higher-level analysis* to determine that one algorithm or data structure is *superior* than others?
2. Comparison of multiple algorithms is only *meaningful* when experiments are conducted under the same environment of:
   - *Hardware*: CPU, running processes
   - *Software*: OS, JVM version
3. Experiments can be done only on *a limited set of test inputs*.
   - What if "*important*" inputs were not included in the experiments?

Moving Beyond Experimental Analysis

- A better approach to analyzing the *efficiency* (e.g., *running times*) of algorithms should be one that:
  - Allows us to calculate the *relative efficiency* (rather than absolute elapsed time) of algorithms in a ways that is *independent of* the hardware and software environment.
  - Can be applied using a *high-level description* of the algorithm (without fully implementing it).
  - Considers *all* possible inputs.
- We will learn a better approach that contains 3 ingredients:
  1. Counting *primitive operations*
  2. Approximating running time as *a function of input size*
  3. Focusing on the *worst-case* input (requiring the most running time)

Counting Primitive Operations

A *primitive operation* corresponds to a low-level instruction with a *constant execution time* .

- Assignment [e.g., `x = 5;`]
- Indexing into an array [e.g., `a[i]`]
- Arithmetic, relational, logical op. [e.g., `a + b`, `z > w`, `b1 && b2`]
- Accessing an attribute of an object [e.g., `acc.balance`]
- Returning from a method [e.g., `return result;`]

  Q: Why is a method call in general *not* a primitive operation?
  A: It may be a call to:
  - a "*cheap*" method (e.g., printing `Hello World`), or
  - an "*expensive*" method (e.g., sorting an array of integers)

Example: Counting Primitive Operations

```
1  findMax (int[] a, int n) {
2    currentMax = a[0];
3    for (int i = 1; i < n; ) {
4      if (a[i] > currentMax) {
5        currentMax = a[i]; }
6      i ++ }
7    return currentMax; }
```

# of times `i < n` in **Line 3** is executed? [n]

# of times the loop body (**Line 4** to **Line 6**) is executed? [$n - 1$]

- **Line 2**: 2 [1 indexing + 1 assignment]
- **Line 3**: $n + 1$ [1 assignment + n comparisons]
- **Line 4**: $(n-1) \cdot 2$ [1 indexing + 1 comparison]
- **Line 5**: $(n-1) \cdot 2$ [1 indexing + 1 assignment]
- **Line 6**: $(n-1) \cdot 2$ [1 addition + 1 assignment]
- **Line 7**: 1 [1 return]
- **Total # of Primitive Operations**: 7n - 2

From Absolute RT to Relative RT

- Each *primitive operation* (PO) takes approximately the same, constant amount of time to execute. [say t]
- The *number of primitive operations* required by an algorithm should be *proportional* to its *actual running time* on a specific environment.
  e.g., `findMax (int[] a, int n)` has $7n - 2$ POs

$$RT = (7n - 2) \cdot t$$

  Say two algorithms with RT $(7n - 2) \cdot t$ and RT $(10n + 3) \cdot t$.
  $\Rightarrow$ It suffices to compare their *relative* running time:
  $7n - 2$ vs. $10n + 3$.
- To determine the *time efficiency* of an algorithm, we only focus on their *number of POs*.

Example: Approx. # of Primitive Operations

- Given # of primitive operations counted precisely as $7n^1 - 2$, we view it as

$$7 \cdot n - 2 \cdot n^0$$

- We say
  - n is the ***highest power***
  - 7 and 2 are the ***multiplicative constants***
  - 2 is the ***lower term***
- When approximating a function (considering that input size may be very large):
  - **Only** the ***highest power*** matters.
  - ***multiplicative constants*** and ***lower terms*** can be dropped.

  $\Rightarrow 7n - 2$ is approximately n

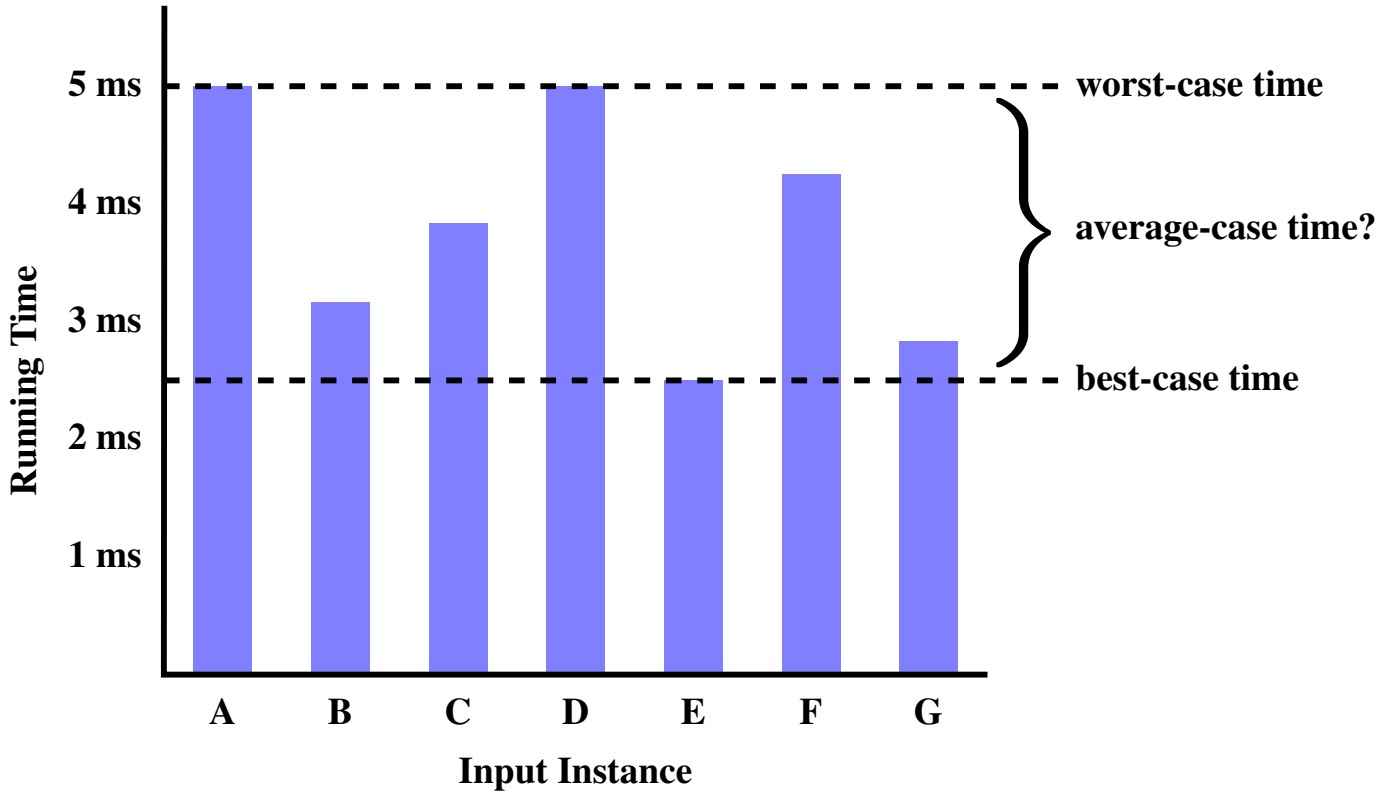
  Exercise: Consider $7n + 2n \cdot log\ n + 3n^2$:
  - ***highest power***? [n^2]
  - ***multiplicative constants***? [7, 2, 3]
  - ***lower terms***? [$7n + 2n \cdot log\ n$]

Approximating Running Time as a Function of Input Size

Given the *high-level description* of an algorithm, we associate it with a function f, such that $f(n)$ returns the *number of primitive operations* that are performed on an *input of size n*.

- $f(n) = 5$ [constant]
- $f(n) = log_2 n$ [logarithmic]
- $f(n) = 4 \cdot n$ [linear]
- $f(n) = n^2$ [quadratic]
- $f(n) = n^3$ [cubic]
- $f(n) = 2^n$ [exponential]

Focusing on the Worst-Case Input



- *Average-case* analysis calculates the *expected running times* based on the probability distribution of input values.
- *worst-case* analysis or *best-case* analysis?

What is Asymptotic Analysis?

Asymptotic analysis

- Is a method of describing *behaviour in the limit*:
  - How the *running time* of the algorithm under analysis changes as the *input size* changes without bound
  - e.g., contrast $RT_1(n) = n$ with $RT_2(n) = n^2$
- Allows us to compare the *relative* performance of alternative algorithms:
  - For large enough inputs, the *multiplicative constants* and *lower-order* terms of an exact running time can be disregarded.
  - e.g., $RT_1(n) = 3n^2 + 7n + 18$ and $RT_1(n) = 100n^2 + 3n - 100$ are considered **equally efficient**, *asymptotically*.
  - e.g., $RT_1(n) = n^3 + 7n + 18$ is considered **less efficient** than $RT_1(n) = 100n^2 + 100n + 2000$, *asymptotically*.

Three Notions of Asymptotic Bounds

We may consider three kinds of *asymptotic bounds* for the *running time* of an algorithm:

- Asymptotic *upper* bound [O]
- Asymptotic lower bound [Ω]
- Asymptotic tight bound [Θ]

Asymptotic Upper Bound: Definition

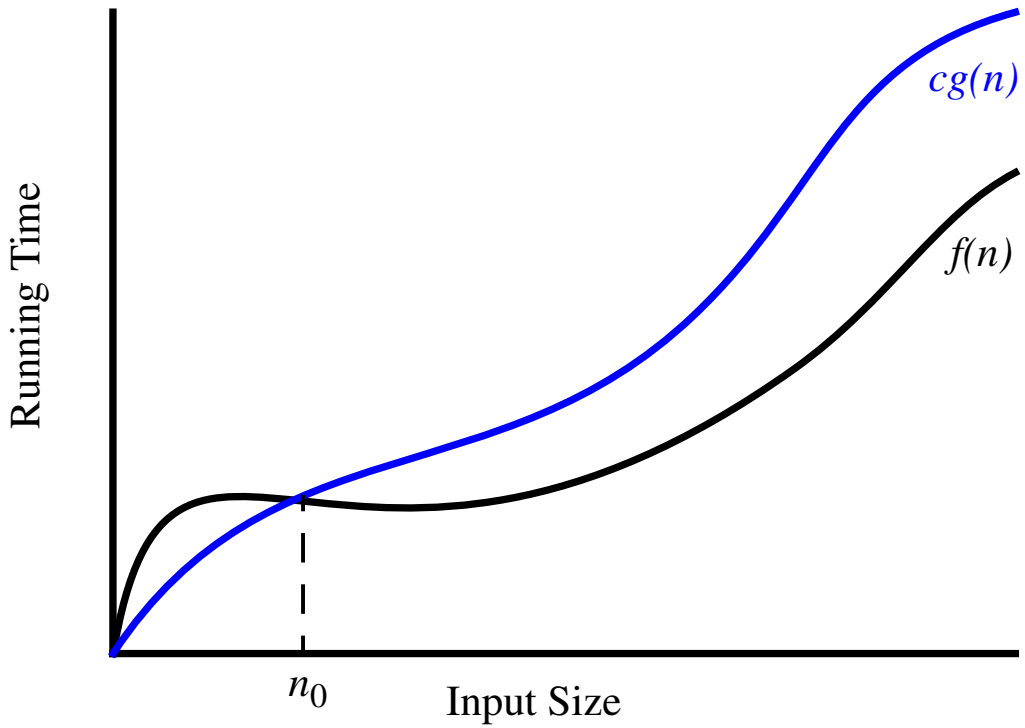
- Let $f(n)$ and $g(n)$ be functions mapping positive integers (input size) to positive real numbers (running time).
  - $f(n)$ characterizes the running time of some algorithm.
  - $O(g(n))$ denotes *a collection of* functions.
- $O(g(n))$ consists of *all* functions that can be upper bounded by $g(n)$, starting at some point, using some constant factor.
- $f(n) \in O(g(n))$ if there are:
  - A real *constant* $c > 0$
  - An integer *constant* $n_0 \geq 1$

  such that:

$$f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$

- For each member function $f(n)$ in $O(g(n))$, we say that:
  - $f(n) \in O(g(n))$ [f(n) is a member of "big-Oh of g(n)"]
  - $f(n)$ **is** $O(g(n))$ [f(n) is "big-Oh of g(n)"]
  - $f(n)$ **is order of** $g(n)$

Asymptotic Upper Bound: Visualization



From n_0, $f(n)$ is upper bounded by $c \cdot g(n)$, so $f(n)$ is $O(g(n))$.

Asymptotic Upper Bound: Example (1)

Prove: The function $8n + 5$ is $O(n)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$8n + 5 \leq c \cdot n$$

Can we choose $c = 9$? What should the corresponding n_0 be?

n	8n + 5	9n
1	13	9
2	21	18
3	29	27
4	37	36
5	45	45
6	53	54

...

Therefore, we prove it by choosing $c = 9$ and $n_0 = 5$.

We may also prove it by choosing $c = 13$ and $n_0 = 1$. Why?

Asymptotic Upper Bound: Example (2)

Prove: The function $f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.

Strategy: Choose a real constant $c > 0$ and an integer constant $n_0 \geq 1$, such that for every integer $n \geq n_0$:

$$5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq c \cdot n^4$$

$f(1) = 5 + 3 + 2 + 4 + 1 = 15$

Choose $c = 15$ and $n_0 = 1$!

Asymptotic Upper Bound: Proposition (1)

If $f(n)$ is a polynomial of degree d, i.e.,

$$f(n) = a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d$$

and $a_0, a_1, \ldots, a_d$ are integers (i.e., negative, zero, or positive), then $f(n)$ *is* $O(n^d)$.

- We prove by choosing

$$\begin{aligned} c &= |a_0| + |a_1| + \cdots + |a_d| \\ n_0 &= 1 \end{aligned}$$

- We know that for $n \geq 1$: $n^0 \leq n^1 \leq n^2 \leq \cdots \leq n^d$
- Upper-bound effect starts when $n_0 = 1$? $[f(1) \leq 1^d]$

$$a_0 \cdot 1^0 + a_1 \cdot 1^1 + \cdots + a_d \cdot 1^d \leq |a_0| \cdot 1^d + |a_1| \cdot 1^d + \cdots + |a_d| \cdot 1^d$$

- Upper-bound effect holds? $[f(n) \leq n^d]$

$$a_0 \cdot n^0 + a_1 \cdot n^1 + \cdots + a_d \cdot n^d \leq |a_0| \cdot n^d + |a_1| \cdot n^d + \cdots + |a_d| \cdot n^d$$

Asymptotic Upper Bound: Proposition (2)

$$O(n^0) \subset O(n^1) \subset O(n^2) \subset \ldots$$

If a function $f(n)$ is *upper bounded* by another function $g(n)$ of degree d, $d \geq 0$, then $f(n)$ is also upper bounded by all other functions of a *strictly higher degree* (i.e., $d+1$, $d+2$, *etc.*).

e.g., Family of $O(n)$ contains:

$n^0, 2n^0, 3n^0, \ldots$	[functions with degree 0]
$n, 2n, 3n, \ldots$	[functions with degree 1]

e.g., Family of $O(n^2)$ contains:

$n^0, 2n^0, 3n^0, \ldots$	[functions with degree 0]
$n, 2n, 3n, \ldots$	[functions with degree 1]
$n^2, 2n^2, 3n^2, \ldots$	[functions with degree 2]

Asymptotic Upper Bound: More Examples

- $5n^2 + 3n \cdot \log n + 2n + 5$ is $O(n^2)$ $[c = 15, n_0 = 1]$
- $20n^3 + 10n \cdot \log n + 5$ is $O(n^3)$ $[c = 35, n_0 = 1]$
- $3 \cdot \log n + 2$ is $O(\log n)$ $[c = 5, n_0 = 2]$
  - Why can't n_0 be 1?
  - Choosing $n_0 = 1$ means $\Rightarrow$ $f(1)$ ***is*** upper-bounded by $c \cdot \log 1$:
    - We have $f(1) = 3 \cdot \log 1 + 2$, which is 2.
    - We have $c \cdot \log 1$, which is 0.

    $\Rightarrow$ $f(1)$ ***is not*** upper-bounded by $c \cdot \log 1$ [Contradiction!]
- 2^{n+2} is $O(2^n)$ $[c = 4, n_0 = 1]$
- $2n + 100 \cdot \log n$ is $O(n)$ $[c = 102, n_0 = 1]$

Using Asymptotic Upper Bound Accurately

- Use the big-Oh notation to characterize a function (of an algorithm's running time) *as closely as possible*.

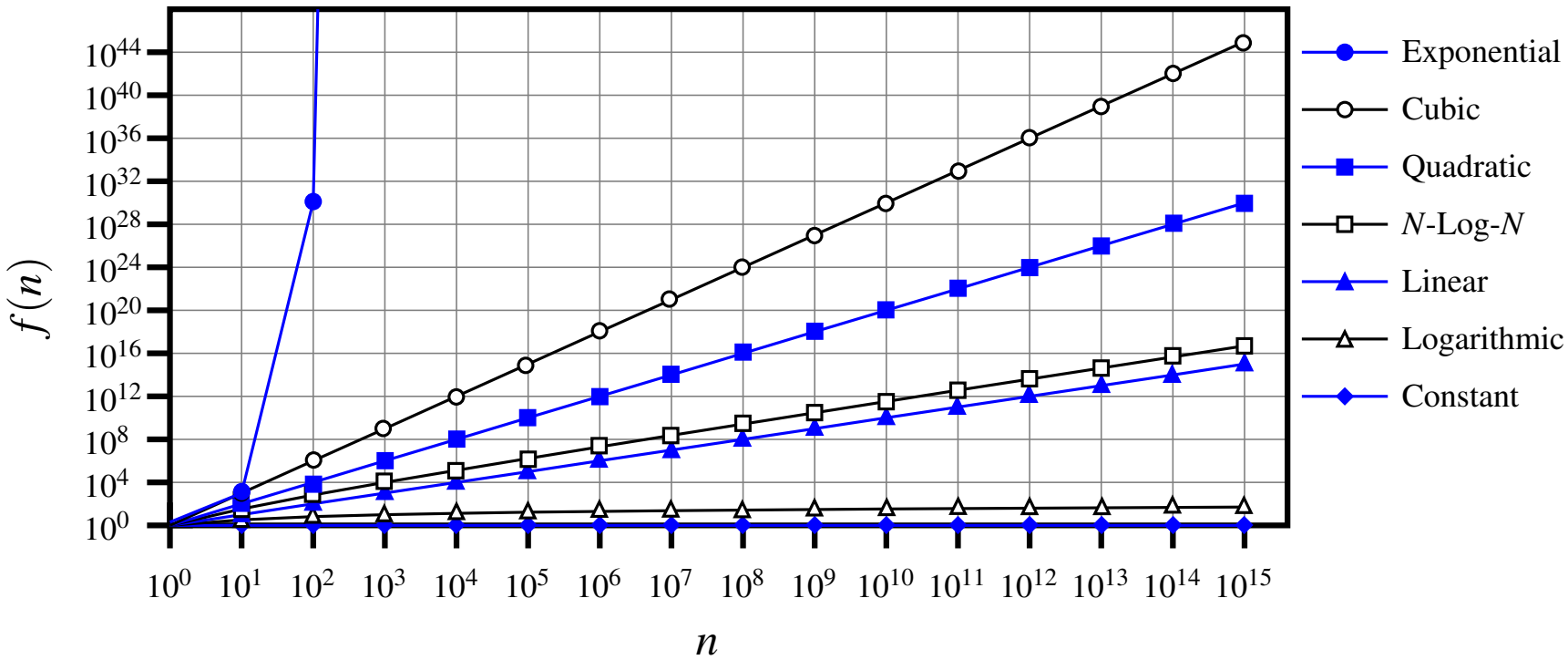
  For example, say $f(n) = 4n^3 + 3n^2 + 5$:
  - Recall: $O(n^3) \subset O(n^4) \subset O(n^5) \subset \ldots$
  - It is the *most accurate* to say that $f(n)$ is $O(n^3)$.
  - It is ***true***, but not very useful, to say that $f(n)$ is $O(n^4)$ and that $f(n)$ is $O(n^5)$.
  - It is ***false*** to say that $f(n)$ is $O(n^2)$, $O(n)$, or $O(1)$.
- Do not include *constant factors* and *lower-order terms* in the big-Oh notation.

  For example, say $f(n) = 2n^2$ is $O(n^2)$, do not say $f(n)$ is $O(4n^2 + 6n + 9)$.

Classes of Functions

upper bound	class	cost
$O(1)$	constant	*cheapest*
$O(log(n))$	logarithmic	
$O(n)$	linear	
$O(n \cdot log(n))$	“n-log-n”	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(n^k), k \geq 1$	polynomial	
$O(a^n), a > 1$	exponential	*most expensive*

Rates of Growth: Comparison



Upper Bound of Algorithm: Example (1)

```
1  maxOf (int x, int y) {
2    int max = x;
3    if (y > x) {
4      max = y;
5    }
6    return max;
7  }
```

- # of primitive operations: 4

  2 assignments + 1 comparison + 1 return = 4

- Therefore, the running time is $O(1)$.
- That is, this is a *constant-time* algorithm.

Upper Bound of Algorithm: Example (2)

```
1  findMax (int[] a, int n) {
2    currentMax = a[0];
3    for (int i = 1; i < n; ) {
4      if (a[i] > currentMax) {
5        currentMax = a[i]; }
6      i ++ }
7    return currentMax; }
```

- From last lecture, we calculated that the # of primitive operations is $7n - 2$.
- Therefore, the running time is $O(n)$.
- That is, this is a *linear-time* algorithm.

Upper Bound of Algorithm: Example (3)

```
1  containsDuplicate (int[] a, int n) {
2    for (int i = 0; i < n; ) {
3      for (int j = 0; j < n; ) {
4        if (i != j && a[i] == a[j]) {
5          return true; }
6        j ++; }
7      i ++; }
8    return false; }
```

- Worst case is when we reach Line 8.
- # of primitive operations $\approx c_1 + n \cdot n \cdot c_2$, where c_1 and c_2 are some constants.
- Therefore, the running time is $O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (4)

```
1   sumMaxAndCrossProducts (int[] a, int n) {
2     int max = a[0];
3     for(int i = 1; i < n;) {
4       if (a[i] > max) { max = a[i]; }
5     }
6     int sum = max;
7     for (int j = 0; j < n; j ++) {
8       for (int k = 0; k < n; k ++) {
9         sum += a[j] * a[k]; } }
10    return sum; }
```

- # of primitive operations $\approx (c_1 \cdot n + c_2) + (c_3 \cdot n \cdot n + c_4)$, where c_1, c_2, c_3, and c_4 are some constants.
- Therefore, the running time is $O(n + n^2) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Upper Bound of Algorithm: Example (5)

```
1  triangularSum (int[] a, int n) {
2    int sum = 0;
3    for (int i = 0; i < n; i ++) {
4      for (int j = i; j < n; j ++) {
5        sum += a[j]; } }
6    return sum; }
```

- # of primitive operations $\approx n + (n-1) + \cdots + 2 + 1 = \frac{n \cdot (n+1)}{2}$
- Therefore, the running time is $O(\frac{n^2+n}{2}) = O(n^2)$.
- That is, this is a *quadratic* algorithm.

Basic Data Structure: Arrays

- An array is a sequence of indexed elements.
- *Size* of an array is **fixed** at the time of its construction.
- Supported *operations* on an array:
  - *Accessing*: e.g., `int max = a[0];`
    Time Complexity: *O(1)* [constant operation]
  - *Updating*: e.g., `a[i] = a[i + 1];`
    Time Complexity: *O(1)* [constant operation]
  - *Inserting/Removing*:

```
String[] insertAt(String[] a, int n, String e, int i)
   String[] result = new String[n + 1];
   for(int j = 0; j <= i - 1; j ++){ result[j] = a[j]; }
   result[i] = e;
   for(int j = i + 1; j <= n - 1; j ++){ result[j] = a[j-1]; }
   return result;
```

Time Complexity: *O(n)* [linear operation]

Array Case Study: Comparing Two Sorting Strategies

- Problem:

  Input: An array a of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

  Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$
- We propose two *alternative implementation strategies* for solving this problem.
- At the end, we want to know which one to choose, based on *time complexity*.

Sorting: Strategy 1 – Selection Sort

- Maintain a (initially empty) *sorted portion* of array *a*.
- From left to right in array *a*, select and insert *the minimum element* to the end of this sorted portion, so it remains sorted.

```
1  selectionSort(int[] a, int n)
2    for (int i = 0; i <= (n - 2); i ++)
3      int minIndex = i;
4      for (int j = i; j <= (n - 1); j ++)
5        if (a[j] < a[minIndex]) { minIndex = j; }
6      int temp = a[i];
7      a[i] = a[minIndex];
8      a[minIndex] = temp;
```

- How many times does the body of *for loop* (Line 4) run?
- Running time? [$O(n^2)$]

$$\underbrace{n}_{\text{find } \{a[0], \ldots, a[n-1]\}} + \underbrace{(n-1)}_{\text{find } \{a[1], \ldots, a[n-1]\}} + \cdots + \underbrace{2}_{\text{find } \{a[n-2], a[a[n-1]]\}}$$

- So selection sort is a *quadratic-time algorithm*.

Sorting: Strategy 2 – Insertion Sort

- Maintain a (initially empty) *sorted portion* of array *a*.
- From left to right in array *a*, insert *one element at a time* into the "right" spot in this sorted portion, so it remains sorted.

```
1  insertionSort(int[] a, int n)
2    for (int i = 1; i < n; i ++)
3      int current = a[i];
4      int j = i;
5      while (j > 0 && a[j - 1] > current)
6        a[j] = a[j - 1];
7        j --;
8      a[j] = current;
```

- *while loop* (L5) exits when? `j <= 0` or `a[j - 1] <= current`
- Running time? [$O(n^2)$]

$$O(\underbrace{1}_{\text{insert into } \{a[0]\}} + \underbrace{2}_{\text{insert into } \{a[0], a[1]\}} + \cdots + \underbrace{(n-1)}_{\text{insert into } \{a[0], \ldots, a[n-2]\}})$$

- So insertion sort is a *quadratic-time algorithm*.

Sorting: Alternative Implementations?

- In the Java implementations for *selection* sort and *insertion* sort, we maintain the "sorted portion" from the *left* end.
  - For *selection* sort, we select the *minimum* element from the "unsorted portion" and insert it to the *end* in the "sorted portion".
- For *insertion* sort, we choose the *left-most* element from the "unsorted portion" and insert it at the "*right spot*" in the "sorted portion".
- **Question:** Can we modify the Java implementations, so that the "sorted portion" is maintained and grown from the *right* end instead?

Comparing Insertion & Selection Sorts

- *Asymptotically*, running times of selection sort and insertion sort are both $O(n^2)$.
- We will later see that there exist better algorithms that can perform better than quadratic: $O(n \cdot logn)$.

Index (1)

Index (2)

Index (3)

Aggregation and Composition

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Call by Value (1)

- Consider the general form of a call to some ***mutator method*** `m1`, with ***context object*** `s` and **argument value** `arg`:

```
class Supplier {
  void m1( T par) {
    /* manipulate par */
  }
}
```

```
class Client {
  Supplier s = new Supplier();
  T arg = ...;
  s.m1(arg)
}
```

  - To execute ***s.m1*(arg)**, an implicit par := **arg** is done.
    ⇒ A *copy* of value stored in **arg** is passed for the method call.

- What can the type *T* be? [Primitive or Reference]
  - *T* is primitive type (e.g., `int`, `char`, `boolean`, *etc.*):
    Call by Value: Copy of **arg**'s *value* (e.g., `2`, `'j'`) is passed.
  - *T* is reference type (e.g., `String`, `Point`, `Person`, *etc.*):
    Call by Value: Copy of **arg**'s *stored reference/address* (e.g., `Point@5cb0d902`) is passed.

Call by Value (2.1)

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

```
class Point {
  int x;
  int y;
  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  void moveVertically(int y){
    this.y += y;
  }
  void moveHorizontally(int x){
    this.x += x;
  }
}
```

Call by Value (2.2.1)

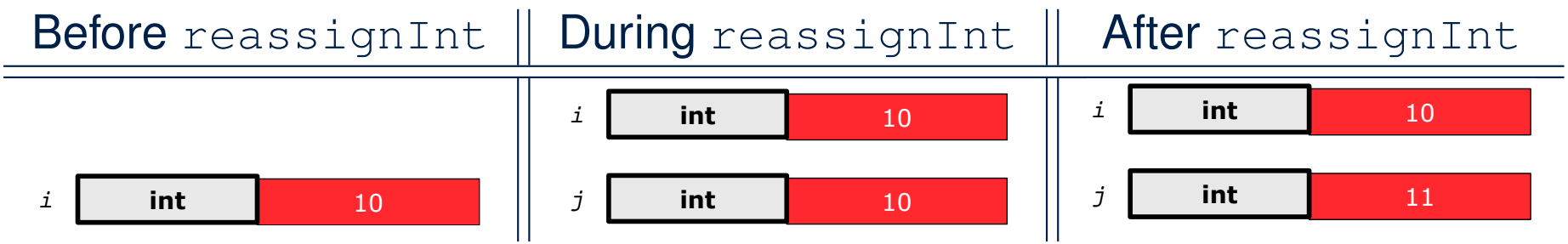
```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
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)



Call by Value (2.3.1)

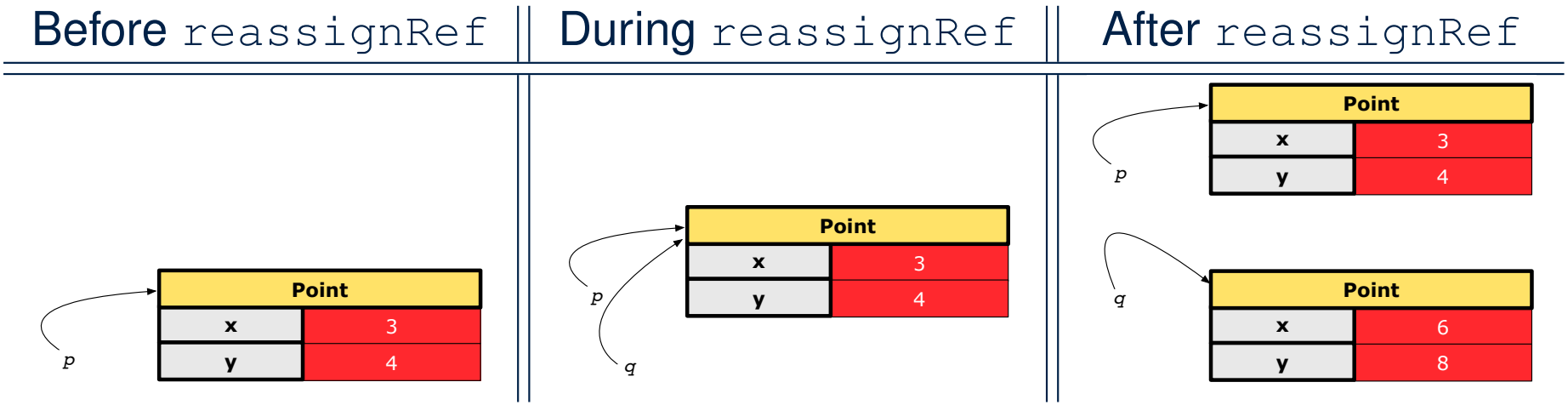
```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
1  @Test
2  public void testCallByRef_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.x==3 && p.y==4);
9  }
```

- ***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)



Call by Value (2.4.1)

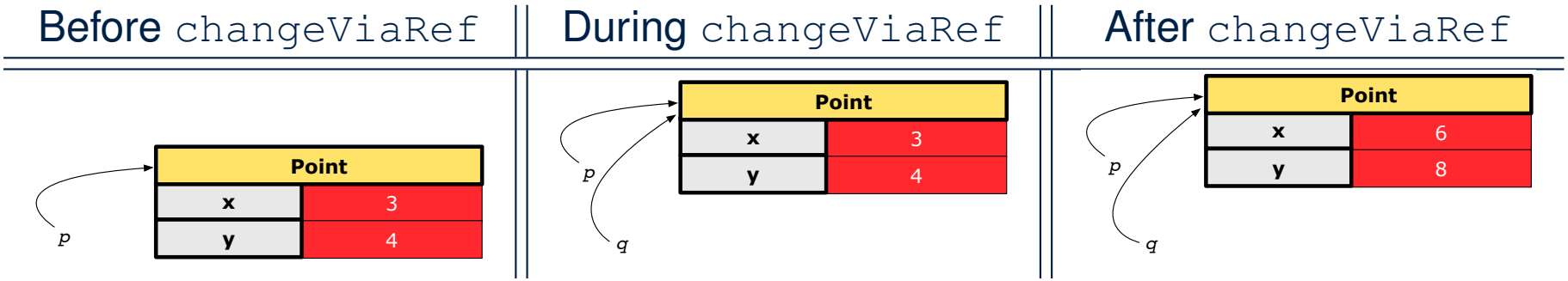
```
public class Util {
  void reassignInt(int j) {
    j = j + 1; }
  void reassignRef(Point q) {
    Point np = new Point(6, 8);
    q = np; }
  void changeViaRef(Point q) {
    q.moveHorizontally(3);
    q.moveVertically(4); } }
```

```
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.x==6 && p.y==8);
9  }
```

- ***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.]

  ⇒ Calls to `q.moveHorizontally` and `q.moveVertically` are effective on both `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` have been modified via `q`.

Call by Value (2.4.2)



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`.

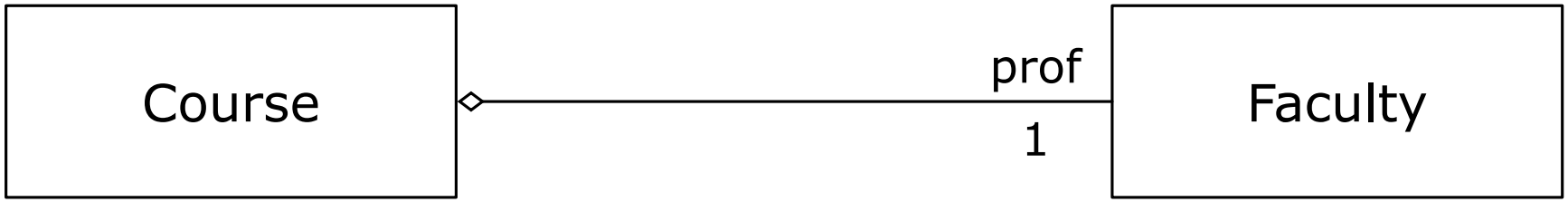
    e.g., Each file has exactly one parent directory.

    ⇒ 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)




```
class Course {
  String title;
  Faculty prof;
  Course(String title) {
    this.title = title;
  }
  void setProf(Faculty prof) {
    this.prof = prof;
  }
  Faculty getProf() {
    return this.prof;
  }
}
```

```
class Faculty {
  String name;
  Faculty(String name) {
    this.name = name;
  }
  void setName(String name) {
    this.name = name;
  }
  String getName() {
    return this.name;
  }
}
```

Aggregation: Independent Containees Shared by Containers (1.2)

```
@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)


```
class Student {
  String id; ArrayList<Course> cs; /* courses */
  Student(String id) { this.id = id; cs = new ArrayList<>(); }
  void addCourse(Course c) { cs.add(c); }
  ArrayList<Course> getCS() { return cs; }
}
```

```
class Course { String title; }
```

```
class Faculty {
  String name; ArrayList<Course> te; /* teaching */
  Faculty(String name) { this.name = name; te = new ArrayList<>(); }
  void addTeaching(Course c) { te.add(c); }
  ArrayList<Course> getTE() { return te; }
}
```

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

The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.


```
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<Course> te;
}
```

Aggregation links between classes constrain how you can *navigate* among these classes.

e.g., In the context of class `Student`:

- Writing ***`cs`*** denotes the list of registered courses.
- Writing ***`cs[i]`*** (where `i` is a valid index) navigates to the class `Course`, which changes the context to class `Course`.

The Dot Notation (3.2)

```
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<Course> te;
}
```

```
class Student {
  ... /* attributes */
  /* Get the student's id */
  String getID() { return this.id; }
  /* Get the title of the ith course */
  String getCourseTitle(int i) {
    return this.cs.get(i).title;
  }
  /* Get the instructor's name of the ith course */
  String getInstructorName(int i) {
    return this.cs.get(i).prof.name;
  }
}
```

The Dot Notation (3.3)

```
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<Course> te;
}
```

```
class Course {
  ... /* attributes */
  /* Get the course's title */
  String getTitle() { return this.title; }
  /* Get the instructor's name */
  String getInstructorName() {
    return this.prof.name;
  }
  /* Get title of ith teaching course of the instructor */
  String getCourseTitleOfInstructor(int i) {
    return this.prof.te.get(i).title;
  }
}
```

The Dot Notation (3.4)

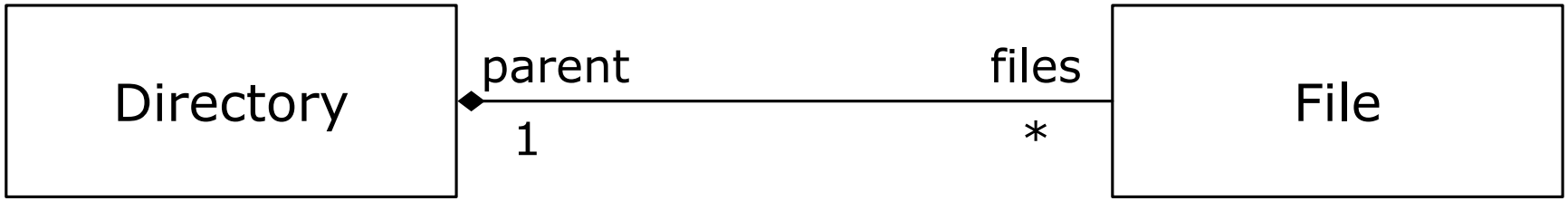
```
class Student {
  String id;
  ArrayList<Course> cs;
}
```

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

```
class Faculty {
  String name;
  ArrayList<Course> te;
}
```

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

Composition: Dependent Containees Owned by Containers (1.1)

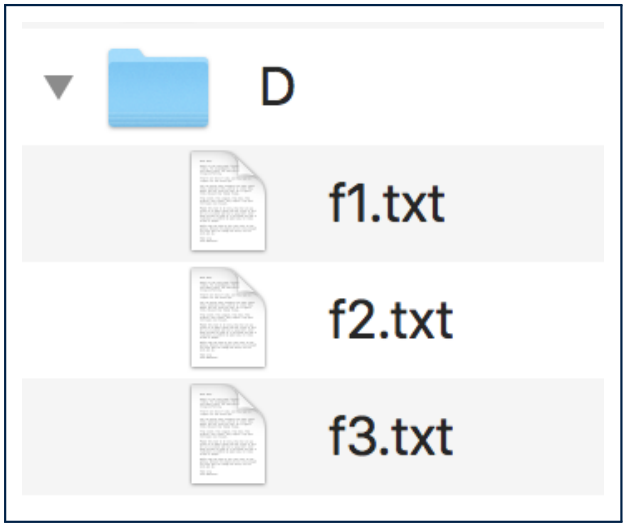


Assumption: Files are not shared among directories.

```
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)

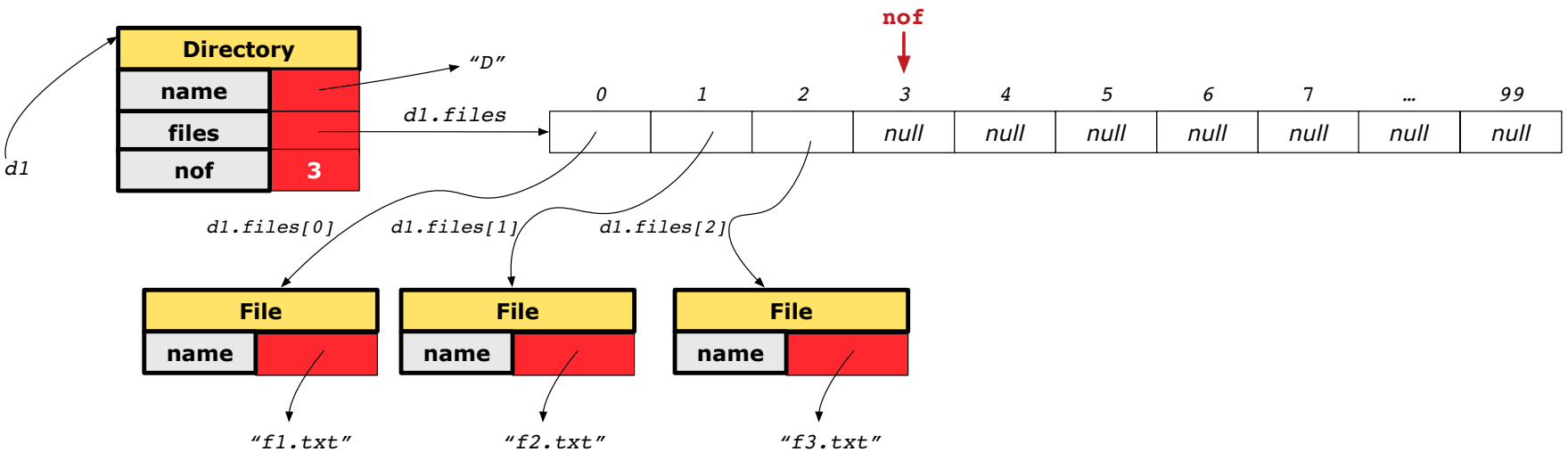



```
1  @Test
2  public void testComposition() {
3    Directory d1 = new Directory("D");
4    d1.addFile("f1.txt");
5    d1.addFile("f2.txt");
6    d1.addFile("f3.txt");
7    assertTrue(
8      d1.files[0].name.equals("f1.txt"));
9  }
```

- **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 *copy constructor* for `Directory`.

A ***copy constructor*** is a constructor which initializes attributes from the argument object `other`.

```
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 =.

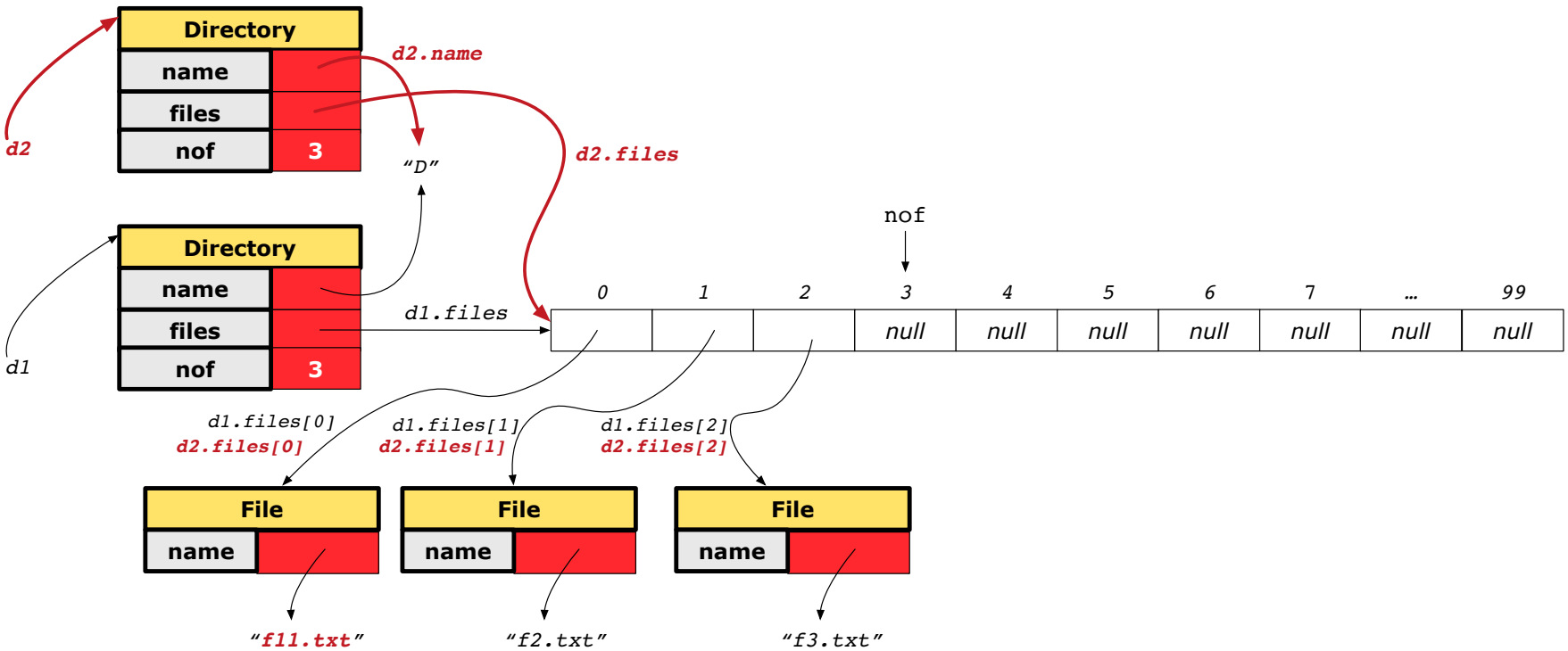
```
class Directory {
  Directory(Directory other) {
    /* value copying for primitive type */
    nof = other.nof;
    /* address copying for reference type */
    name = other.name; files = other.files; } }
```

Is a shallow copy satisfactory to support composition?
i.e., Does it still forbid sharing to occur? [***NO***]

```
@Test
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.files == d2.files); /* violation of composition */
  d2.files[0].changeName("f11.txt");
  assertFalse(d1.files[0].name.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*

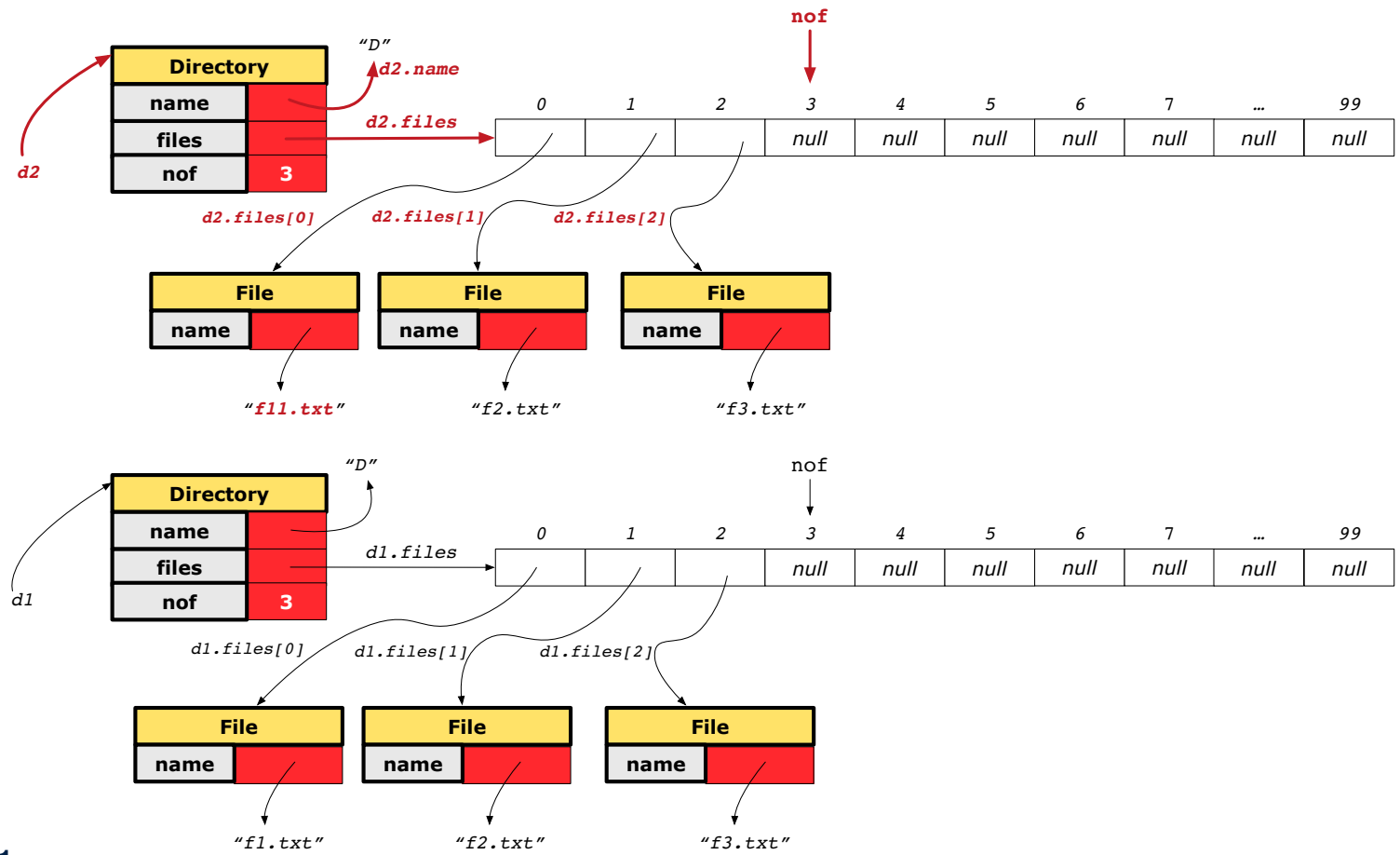
```
class File {
  File(File other) {
    this.name =
      new String(other.name);
  }
}
```

```
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 < nof; i ++) {
      File src = other.files[i];
      File nf = new File(src);
      this.addFile(nf); } }
  void addFile(File f) { ... } }
```

```
@Test
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.files != d2.files); /* composition preserved */
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0].name.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 < nof; i ++) {
      File src = other.files[i];
      this.addFile(src); } }
  void addFile(File f) { ... } }
```

```
@Test
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.files != d2.files); /* composition preserved */
  d2.files[0].changeName("f11.txt");
  assertTrue(d1.files[0] == d2.files[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.

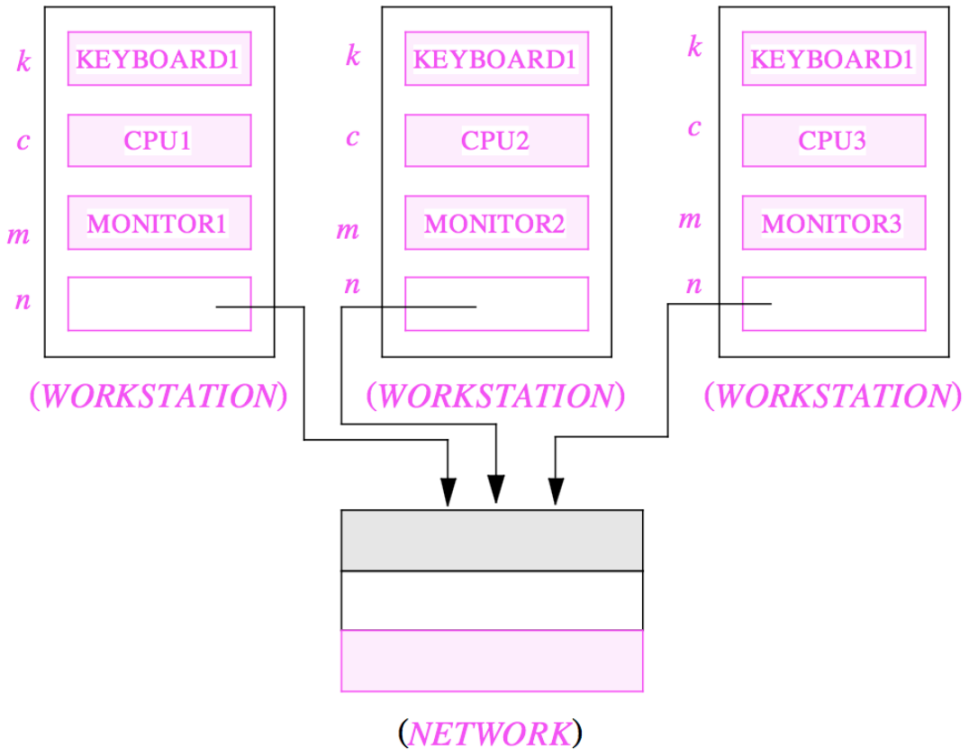
Composition:

- 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***]



Index (1)

Index (2)

Index (3)

Inheritance

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

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

```
class ResidentStudent {
  String name;
  Course[] registeredCourses;
  int numberOfCourses;
  double premiumRate; /* there's a mutator method for this */
  ResidentStudent (String name) {
    this.name = name;
    registeredCourses = new Course[10];
  }
  void register(Course c) {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
  double getTuition() {
    double tuition = 0;
    for(int i = 0; i < numberOfCourses; i ++) {
      tuition += registeredCourses[i].fee;
    }
    return tuition * premiumRate;
  }
}
```

No Inheritance: `NonResidentStudent` Class

```
class NonResidentStudent {
  String name;
  Course[] registeredCourses;
  int numberOfCourses;
  double discountRate; /* there's a mutator method for this */
  NonResidentStudent (String name) {
    this.name = name;
    registeredCourses = new Course[10];
  }
  void register(Course c) {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
  double getTuition() {
    double tuition = 0;
    for(int i = 0; i < numberOfCourses; i ++) {
      tuition += registeredCourses[i].fee;
    }
    return tuition * discountRate;
  }
}
```

No Inheritance: Testing Student Classes

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

```
class StudentTester {
  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());
  }
}
```

No Inheritance: Issues with the Student Classes

- Implementations for the two student classes seem to work. But can you see any potential problems with it?
- 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*.

No Inheritance: Maintainability of Code (1)

What if the way for registering a course changes?

e.g.,

```
void register(Course c) {
  if (numberOfCourses >= MAX_ALLOWANCE) {
    throw new IllegalArgumentException("Maximum allowance reached.");
  }
  else {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
}
```

We need to change the `register` method in *both* student classes!

No Inheritance: Maintainability of Code (2)

What if the way for calculating the base tuition changes?

e.g.,

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

We need to change the `getTuition` method in *both* student classes.

No Inheritance: A Collection of Various Kinds of Students

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

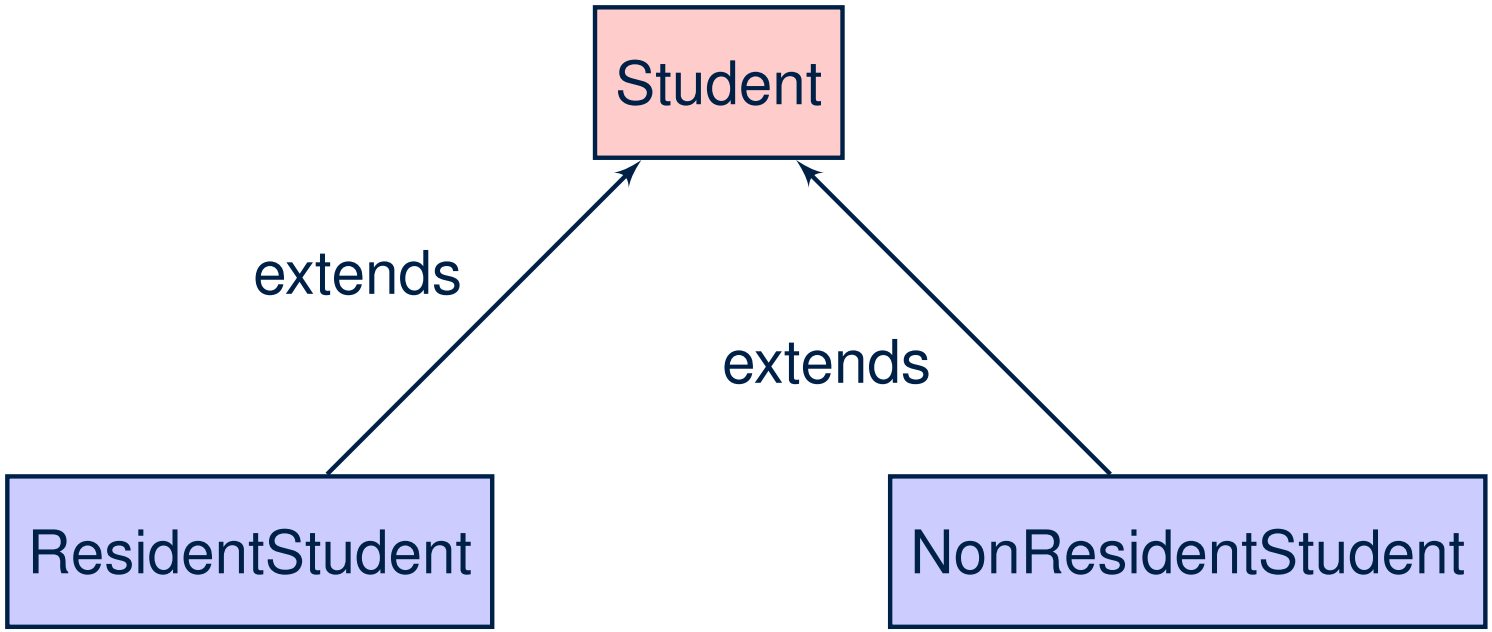
```
class StudentManagementSystem {
  ResidentStudent[] rss;
  NonResidentStudent[] nrss;
  int nors; /* number of resident students */
  int nonrs; /* number of non-resident students */
  void addRS (ResidentStudent rs){ rss[nors]=rs; nors++; }
  void addNRS (NonResidentStudent nrs){ nrss[nonrs]=nrs; nonrs++; }
  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

Inheritance Architecture



Inheritance: The `Student` Parent/Super Class

```
class Student {
  String name;
  Course[] registeredCourses;
  int numberOfCourses;
  Student(String name) {
    this.name = name;
    registeredCourses = new Course[10];
  }
  void register(Course c) {
    registeredCourses[numberOfCourses] = c;
    numberOfCourses ++;
  }
  double getTuition() {
    double tuition = 0;
    for(int i = 0; i < numberOfCourses; i ++) {
      tuition += registeredCourses[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  }
```

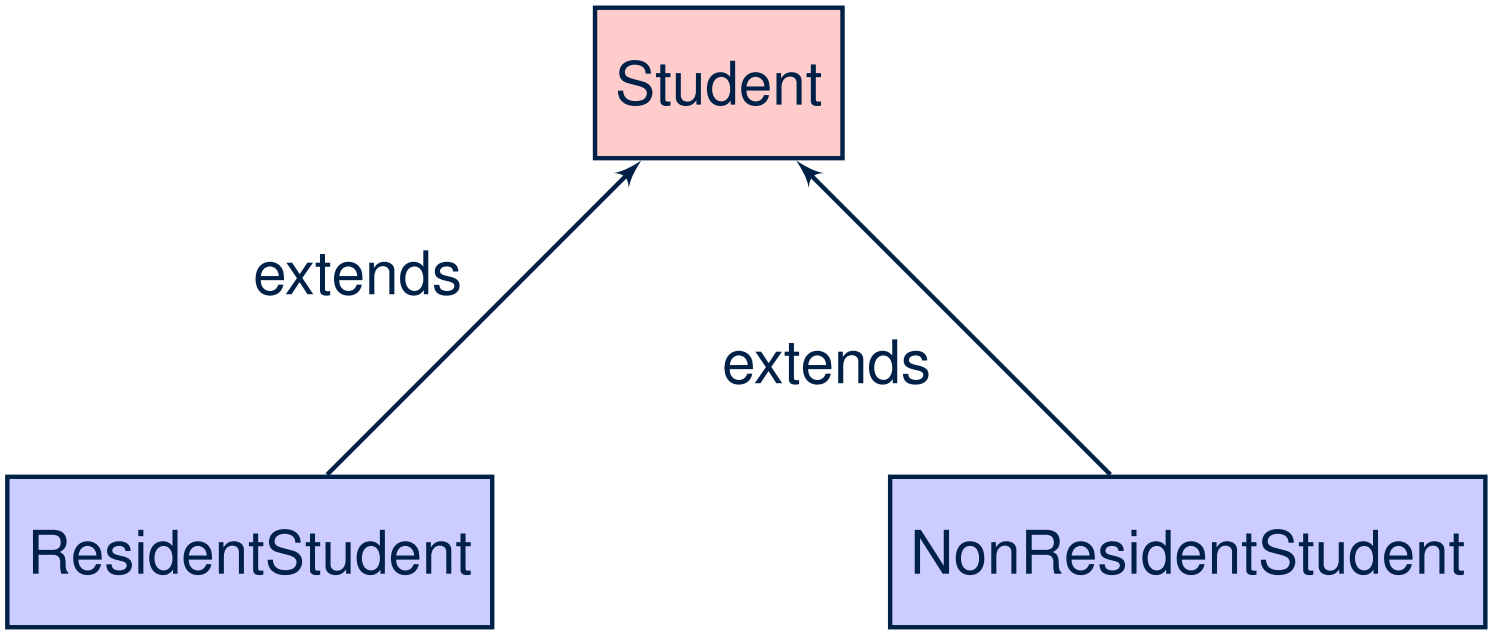
- **L1** declares that `ResidentStudent` inherits all attributes and methods (except constructors) from `Student`.
- There is no need to repeat the `register` method
- Use of *super* in **L4** is as if calling `Student(name)`
- Use of *super* in **L8** returns what `getTuition()` in `Student` returns.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

Inheritance Architecture Revisited



- 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`, `registeredCourses`, `numberOfCourses`
    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`

Visualizing Parent/Child Objects (1)

- A child class inherits ***all*** attributes from its parent class.

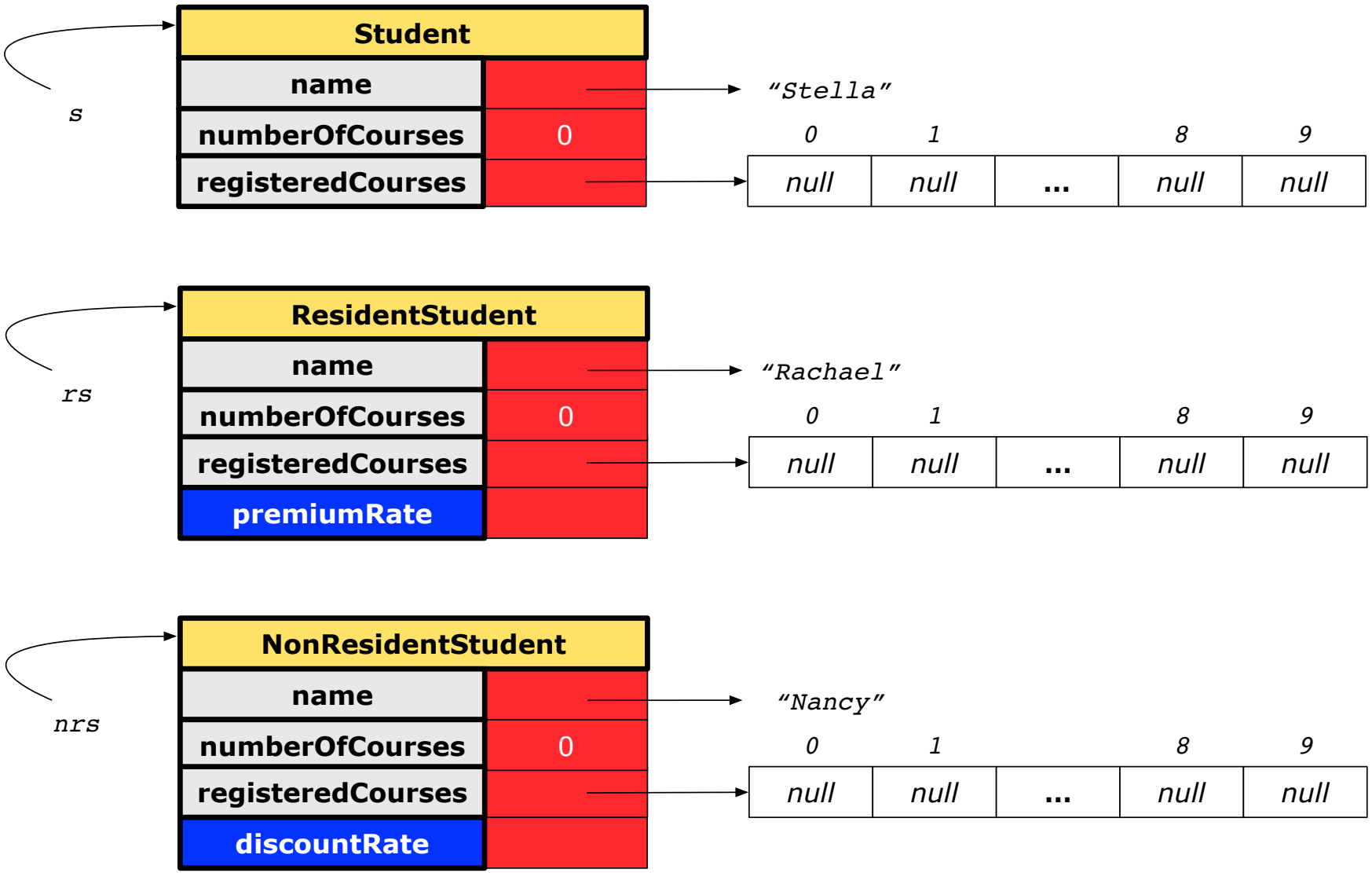
  ⇒ A child instance has ***at least as many*** attributes as an instance of its parent class.

  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?

Visualizing Parent/Child Objects (2)

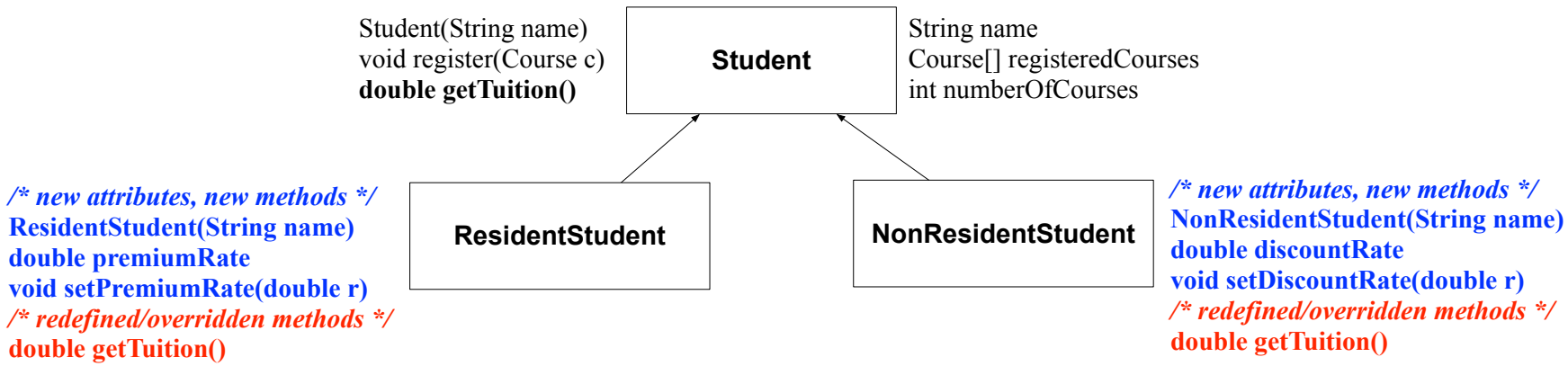


Testing the Two Student Sub-Classes

```
class StudentTester {
  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 exactly the same way as before (because we did not modify *method signatures*).
- But now the internal structure of code has been made *maintainable* using *inheritance*.

Inheritance Architecture: Static Types & Expectations




```
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`]

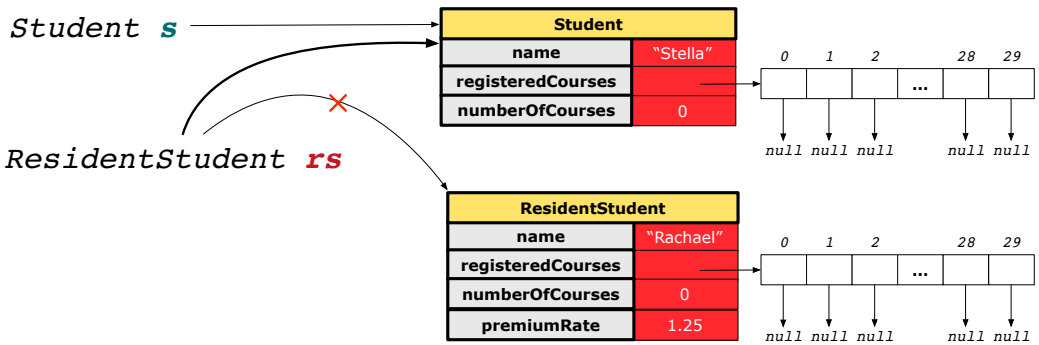
    ∴ The context object `s` is *expected* to be used as:
    - `s.register(eecs2030)` and `s.getTuition()`
  - **L2**: What *kind* of address can `rs` store? [`ResidentStudent`]

    ∴ 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*:



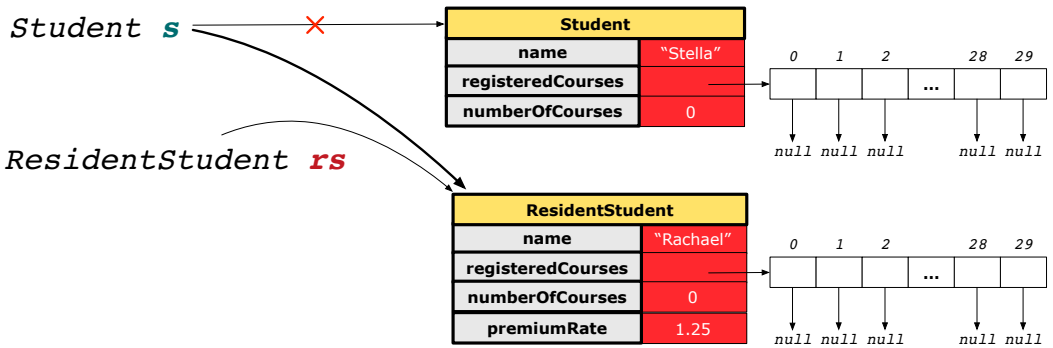
- Since **rs** is declared of type `ResidentStudent`, a subsequent call **rs**`.setPremiumRate(1.50)` can be expected.
- **rs** is now pointing to a `Student` object.
- Then, what would happen to **rs**`.setPremiumRate(1.50)`?

  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()`?

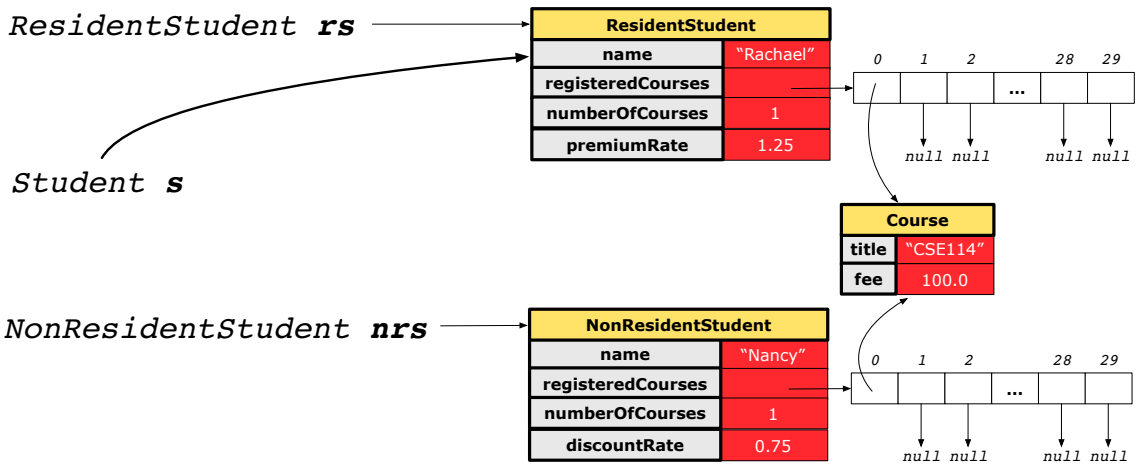
OK ∵ **s**`.premiumRate` is *never directly used*!!

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());/* output: 125.0 */
8  s = nrs;  System.out.println( s .getTuition());/* output: 75.0 */
```

After `s = rs` (**L7**), `s` points to a `ResidentStudent` object.

⇒ Calling `s.getTuition()` applies the `premiumRate`.

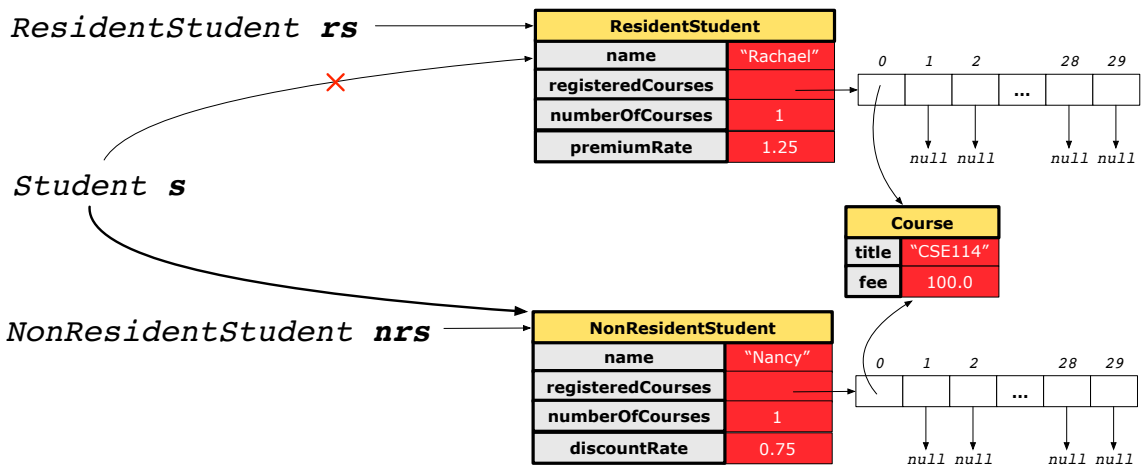


Dynamic Binding: Intuition (2)

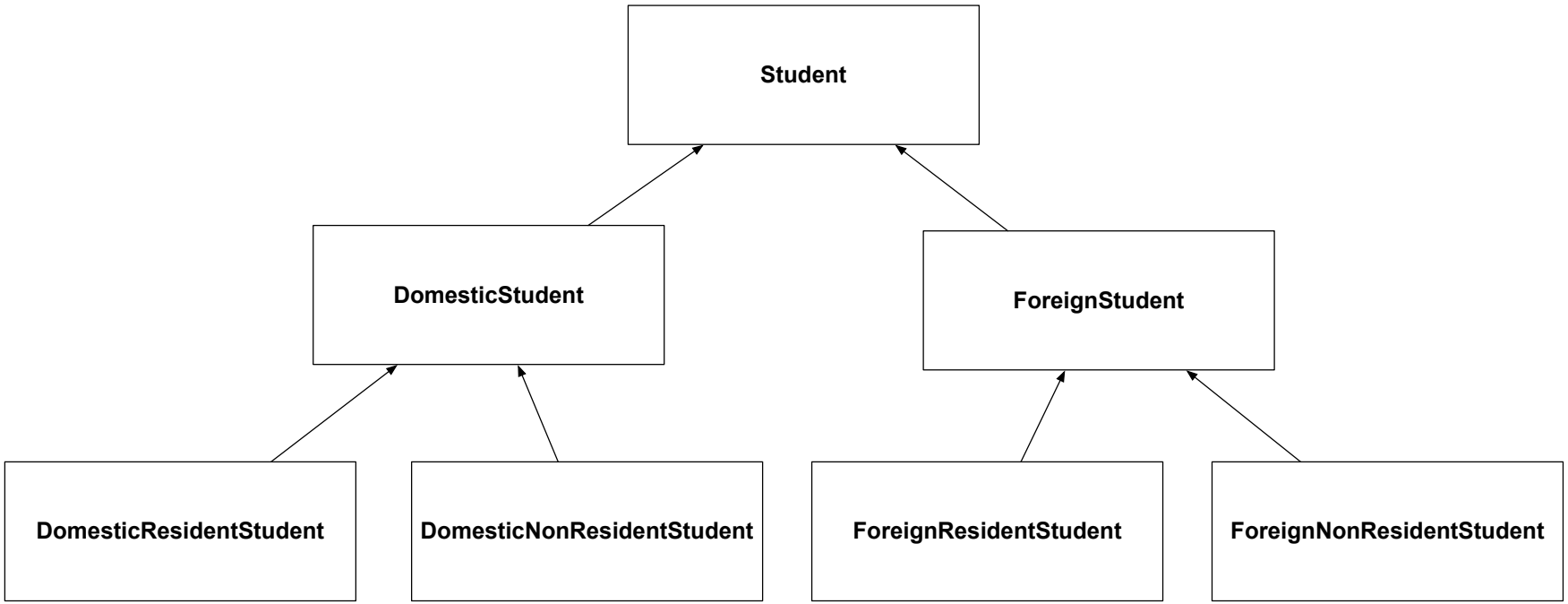
```
1  Course eecs2030 = new Course("EECS2030", 100.0);
2  Student s;
3  ResidentStudent rs = new ResidentStudent("Rachael");
4  NonResidentStudent nrs = new NonResidentStudent("Nancy");
5  rs.setPremiumRate(1.25); rs.register(eecs2030);
6  nrs.setDiscountRate(0.75); nrs.register(eecs2030);
7  s = rs;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

After `s = nrs` (**L8**), `s` points to a `NonResidentStudent` object.

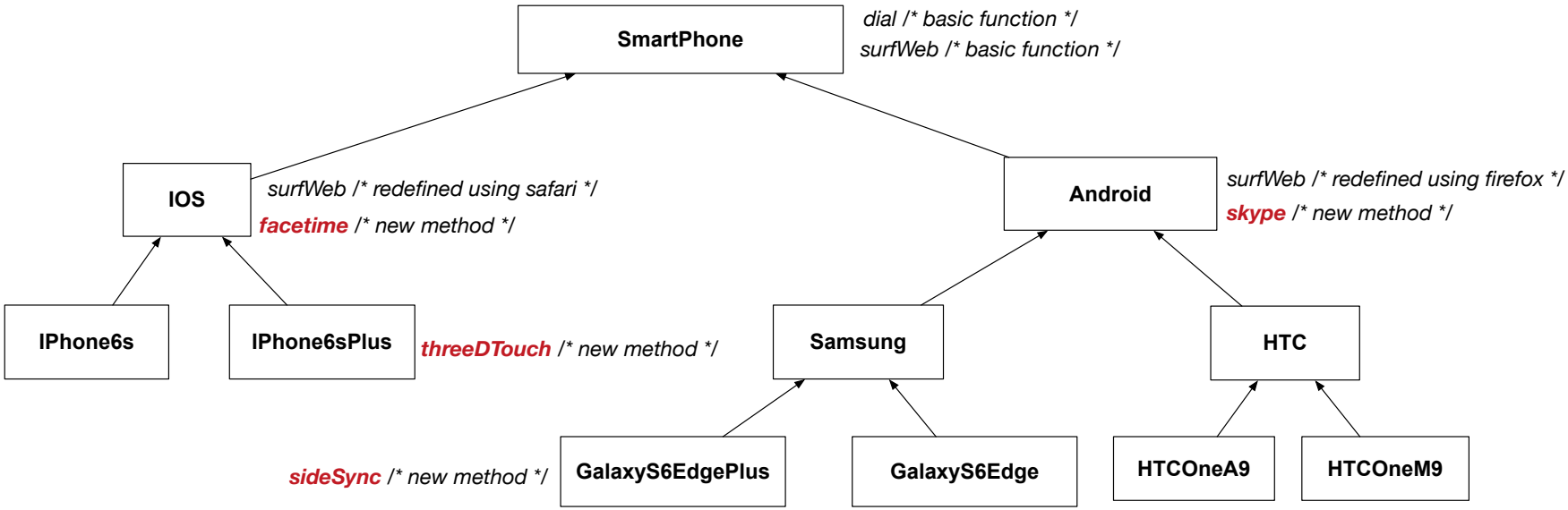
⇒ Calling `s.getTuition()` applies the `discountRate`.



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` inherits 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 inherited to all its *descendant classes*.

Inheritance Accumulates Code for Reuse

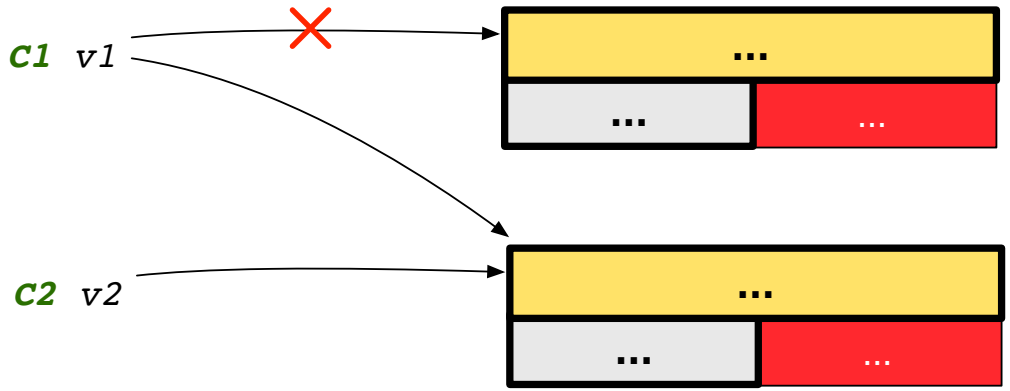
- The *lower* a class is in the type hierarchy, the *more code* it accumulates from its *ancestor classes*:
  - A *descendant class* inherits all code from its *ancestor classes*.
  - A *descendant class* may also:
    - Declare new attributes
    - Define new methods
    - *Redefine* / *Override* inherited 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).
  - When expecting 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 `Student` object, we may substitute it with either a `ResidentStudent` or a `NonResidentStudent` object.
  - **Justification**: A *descendant class* contains *at least as many* methods as defined in its *ancestor classes* (but not vice versa!).

Reference Variable: Static Type

- 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 `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` by an object of (*static*) type `C2`.
- *Substitutions* are subject to *rules*!

Rules of Substitution

When expecting an object of ***static type*** `A`:

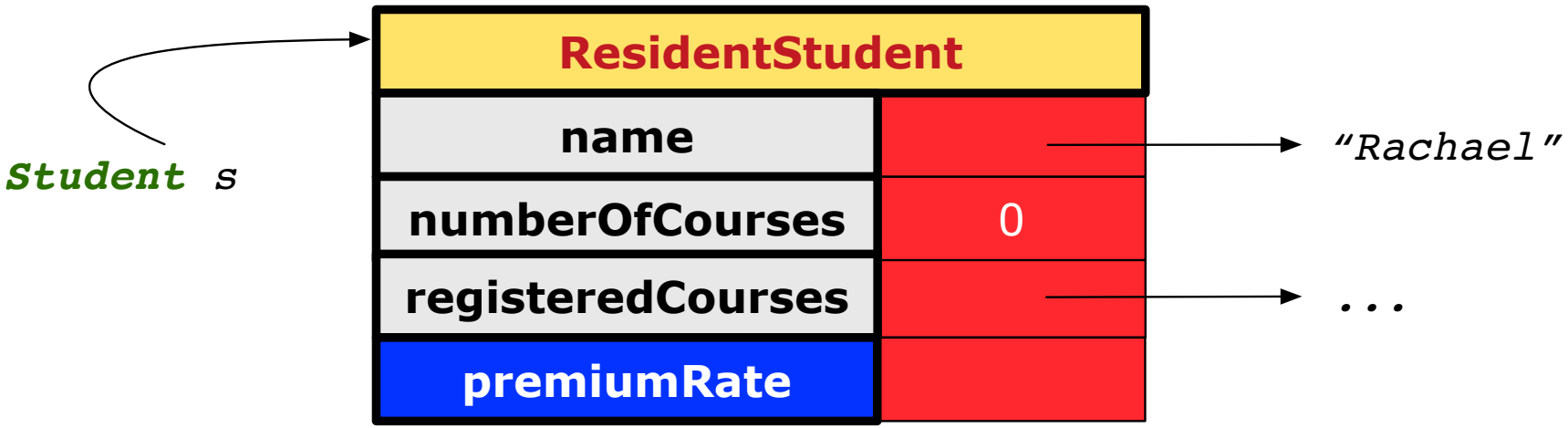
- It is *safe* to *substitute* it with an object whose ***static type*** is any of the *descendant class* of `A` (including `A`).
  - ∵ Each *descendant class* of `A`, being the new substitute, is guaranteed to contain all (non-private) attributes/methods defined in `A`.
  - e.g., When expecting an `IOS` phone, you *can* substitute it with either an `IPhone6s` or `IPhone6sPlus`.
- 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 neither an ancestor nor a descendant of `A`.
  - e.g., When expecting `IOS` phone, *unsafe* to substitute it with an `HTC` ∵ `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 runtime.

- 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***).

Reference Variable: Changing Dynamic Type (1)

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., `Student jim = new NonResidentStudent(...)` changes the *dynamic type* of `jim` to `NonResidentStudent`.
- e.g., `ResidentStudent jim = new Student(...)` is illegal because `Studnet` is **not** a *descendant class* of the *static type* of `jim` (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., `v = other`):

- Substitution Principle: the static type of `other` must be a *descendant class* of `v`'s *static type*.
- e.g., Say we declare

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

- `rs = jim` ×
- `nrs = jim` ×
- `jim = rs` ✓
  changes the *dynamic type* of `jim` to the dynamic type of `rs`
- `jim = nrs` ✓
  changes the *dynamic type* of `jim` to the dynamic type of `nrs`

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 {...}
```

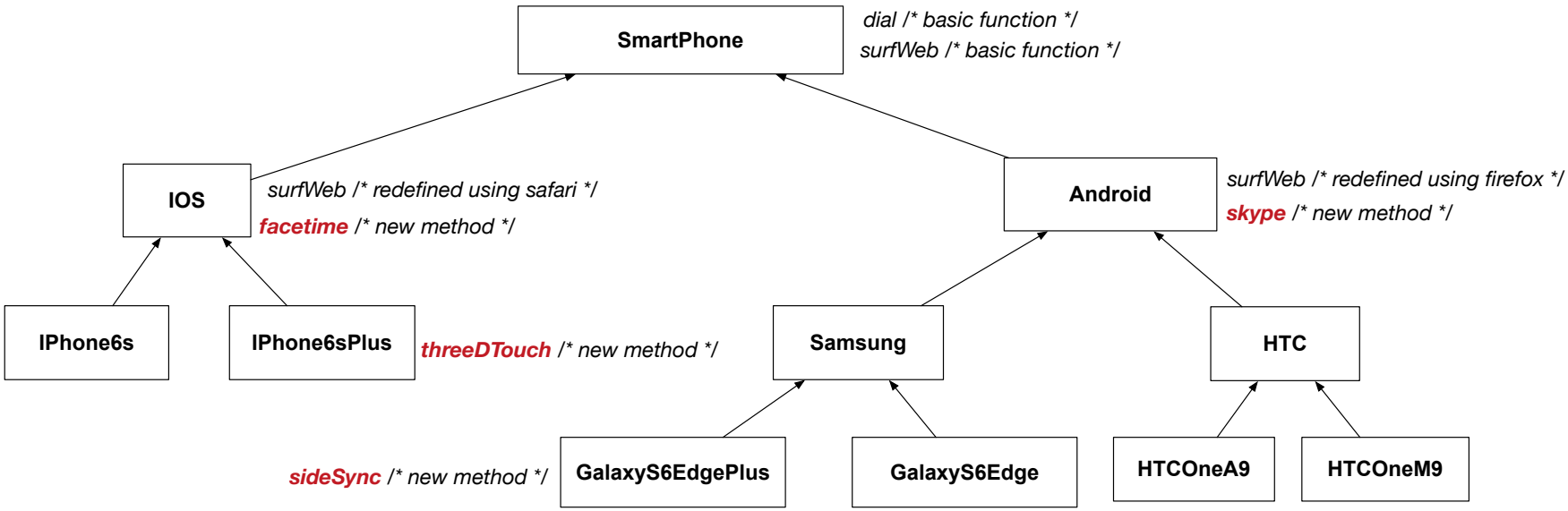
```
class StudentTester1 {
  public static void main(String[] args) {
    Student jim = new Student("J. Davis");
    ResidentStudent rs = new ResidentStudent("J. Davis");
    jim = rs;  /* legal */
    rs = jim;  /* illegal */

    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 */
  }
}
```

Polymorphism and Dynamic Binding (3.1)



Polymorphism and Dynamic Binding (3.2)

```
class SmartPhoneTest1 {
  public static void main(String[] args) {
    SmartPhone myPhone;
    IOS ip = new IPhone6sPlus();
    Samsung ss = new GalaxyS6Edge();
    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 IPhone6sPlus();
    myPhone = ip;
    myPhone. surfWeb ();   /* version of surfWeb in IPhone6sPlus */

    Samsung ss = new GalaxyS6Edge();
    myPhone = ss;
    myPhone. surfWeb ();   /* version of surfWeb in GalaxyS6Edge */
  }
}
```

Reference Type Casting: Motivation (1.1)

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

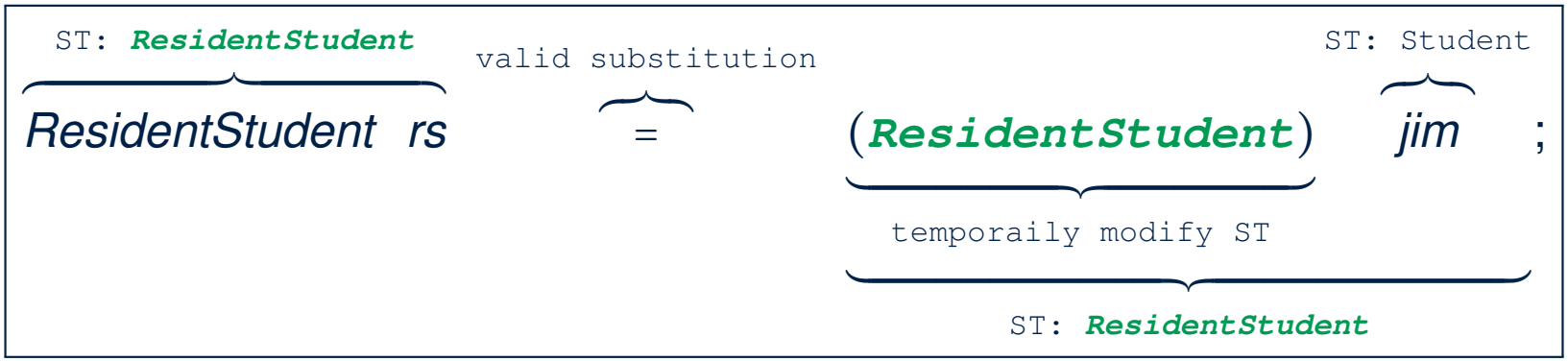
- **L1** is *legal*: `ResidentStudent` is a **descendant class** of the ***static type*** of `jim` (i.e., `Student`).
- **L2** is *illegal*: `jim`'s ***ST*** (i.e., `Student`) is ***not*** a **descendant class** of `rs`'s ***ST*** (i.e., `ResidentStudent`).

  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` on the **RHS of =** temporarily modifies `jim`'s ***ST*** to `ResidentStudent`.
  - Alias `rs` of ***ST*** `ResidentStudent` is then created via an assignment.
- *dynamic binding*: After the *cast*, **L3** will execute the correct version of `setPremiumRate`.

Reference Type Casting: Motivation (1.2)



- Variable `rs` is declared of ***static type*** (***ST***) `ResidentStudent`.
- Variable `jim` is declared of ***ST*** `Student`.
- The cast expression (***ResidentStudent***) jim *temporarily* modifies `jim`'s ***ST*** to `ResidentStudent`.
  - ⇒ Such a cast makes the assignment valid.
  - ∵ RHS's ***ST*** (`ResidentStudent`) is a descendant of LHS's ***ST*** (`ResidentStudent`).
  - ⇒ The assignment creates an alias `rs` with ***ST*** `ResidentStudent`.
- **No** new object is created.
  - Only an *alias* `rs` with a different ***ST*** (`ResidentStudent`) is created.
- After the assignment, `jim`'s ***ST*** *remains* `Student`.

Reference Type Casting: Motivation (2.1)

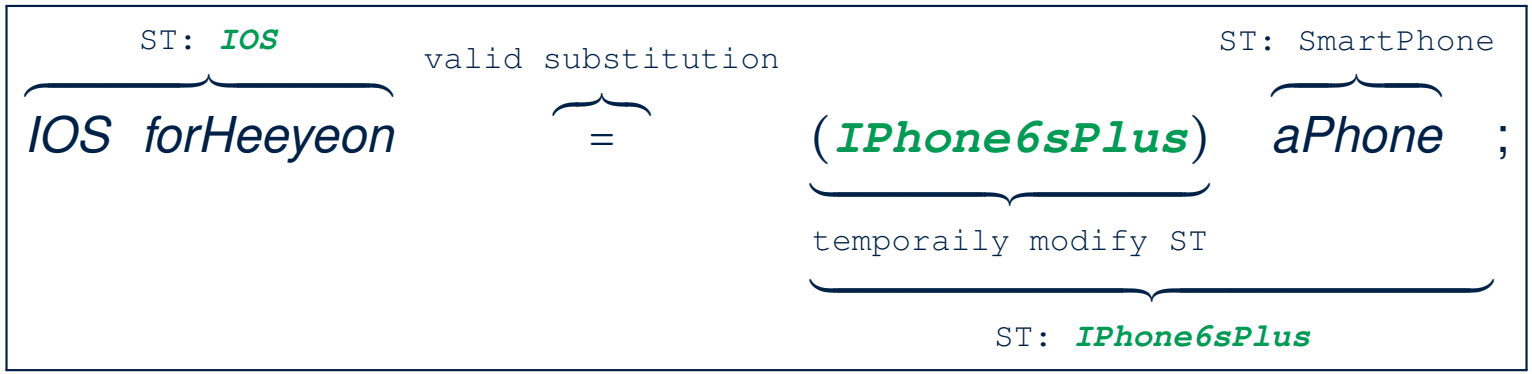
```
1  SmartPhone aPhone = new IPhone6sPlus();
2  IOS forHeeyeon = aPhone;
3  forHeeyeon.facetime();
```

- **L1** is *legal*: `IPhone6sPlus` 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., `IOS`).
  Java compiler is *unable to infer* that `aPhone`'s ***dynamic type*** in **L2** is `IPhone6sPlus`!
- Force Java compiler to believe so via a *cast* in **L2**:

```
IOS forHeeyeon = (IPhone6sPlus) aPhone;
```

  - The cast `(IPhone6sPlus) aPhone` on the **RHS of =** temporarily modifies `aPhone`'s ***ST*** to `IPhone6sPlus`.
  - Alias `forHeeyeon` of ***ST*** `IOS` is then created via an assignment.
- *dynamic binding*: After the *cast*, **L3** will execute the correct version of `facetime`.

Reference Type Casting: Motivation (2.2)



- Variable `forHeeyeon` is declared of ***static type*** (***ST***) `IOS`.
- Variable `aPhone` is declared of ***ST*** `SmartPhone`.
- The cast expression (***IPhone6sPlus***) aPhone *temporarily* modifies `aPhone`'s ***ST*** to `IPhone6sPlus`.
  - ⇒ Such a cast makes the assignment valid.
  - ∵ RHS's ***ST*** (`IPhone6sPlus`) is a descendant of LHS's ***ST*** (`IOS`).
  - ⇒ The assignment creates an alias `forHeeyeon` with ***ST*** `IOS`.
- **No** new object is created.

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

Type Cast: Named or Anonymous

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

```
SmartPhone aPhone = new IPhone6sPlus();
IOS forHeeyeon = (IPhone6sPlus) aPhone;
forHeeyeon.facetime();
```

Anonymous Cast: Use the cast result directly.

```
SmartPhone aPhone = new IPhone6sPlus();
((IPhone6sPlus) aPhone).facetime();
```

Common Mistake:

```
1 SmartPhone aPhone = new IPhone6sPlus();
2 (IPhone6sPlus) aPhone.facetime();
```

L2 ≡ `(IPhone6sPlus) (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* **temporarily** changes the ***ST*** of v from ST_v to *C*.
  ⇒ All methods that are defined in *C* can be called.

```
Android myPhone = new GalaxyS6EdgePlus();
/* 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  × */
GalaxyS6EdgePlus ga = (GalaxyS6EdgePlus) myPhone;
/* Compiles OK ∵ GalaxyS6EdgePlus is a descendant class of Android
 * expectations on ga widened to methods in GalaxyS6EdgePlus
 * 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 GalaxyS6EdgePlus();
2  IPhone6sPlus forHeeyeon = (IPhone6sPlus) aPhone;
3  forHeeyeon.threeDTouch();
```

- **L1** is *legal*: `GalaxyS6EdgePlus` 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 (`IPhone6sPlus`).
- **L3** is *legal* ∵ `threeDTouch` is in `forHeeyeon`' ***ST*** `IPhone6sPlus`.
- Java compiler is *unable to infer* that `aPhone`'s *dynamic type* in **L2** is actually `NonResidentStudent`.
- Executing **L2** will result in a *`ClassCastException`*.
  ∵ Methods `facetime`, `threeDTouch` (expected from an ***IPhone6sPlus***) is *undefined* on the ***GalaxyS6EdgePlus*** 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  GalaxyS6EdgePlus ga = (GalaxyS6EdgePlus) myPhone;
4  /* Compiles OK ∵ GalaxyS6EdgePlus is a descendant class of SmartPhone
5   * can now call methods declared in GalaxyS6EdgePlus 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* `GalaxyS6EdgePlus` (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  IPhone6sPlus ip = (IPhone6sPlus) myPhone;
4  /* Compiles OK ∵ IPhone6sPlus is a descendant class of SmartPhone
5   * can now call methods declared in IPhone6sPlus on ip
6   * ip.dial, ip.surfweb, ip.facetime, ip.threeDTouch  ✓ */
```

- Type cast in **L3** is *compilable*.
- Executing **L3** will cause *`ClassCastException`*.

  L3: `myPhone`'s *DT* `Samsung` cannot meet expectations of the temporary *ST* `IPhone6sPlus` (e.g., `threeDTouch`).

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., `GalaxyS6EdgePlus`) will cause `ClassCastException`.
- Casting `myPhone` to a class irrelevant to its ***DT Samsung*** (e.g., `HTCOneA9`) will cause `ClassCastException`.

Required Reading: Static Types, Dynamic Types, Casts

https://www.eecs.yorku.ca/~jackie/teaching/lectures/2018/F/EECS2030/notes/EECS2030_F18_Notes_Static_Types_Cast.pdf

Compilable Cast vs. Exception-Free Cast

```
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 *dynamic type* is `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 ***static type*** (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 GalaxyS6EdgePlus();
2  if (aPhone instanceof IPhone6sPlus) {
3    IOS forHeeyeon = (IPhone6sPlus) aPhone;
4    forHeeyeon.facetime();
5  }
```

- **L1** is *legal*: `GalaxyS6EdgePlus` is a **descendant class** of the static type of `aPhone` (i.e., `SmartPhone`).
- **L2** checks if `aPhone`'s *dynamic type* is `IPhone6sPlus`.

  FALSE ∵ `aPhone`'s *dynamic type* is `GalaxyS6EdgePlus`!
- **L3** is *legal*: `aPhone`'s cast type (i.e., `IPhone6sPlus`) 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 instanceof C`

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

```
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 GalaxyS6Edge);
/* false ∵ Samsung is not a descendant of GalaxyS6Edge */
println(myPhone instanceof IOS);
/* false ∵ Samsung is not a descendant of IOS */
println(myPhone instanceof IPhone6sPlus);
/* false ∵ Samsung is not a descendant of IPhone6sPlus */
```

⇒ ***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 GalaxyS6EdgePlus) {
7    GalaxyS6EdgePlus galaxy = (GalaxyS6EdgePlus) myPhone;
8  }
9  if(myphone instanceof HTC) {
10   HTC htc = (HTC) 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 Type and Polymorphism (1.1)

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone6sPlus extends IOS {
  void threeDTouch() { ... }
}
```

```
1  SmartPhone sp = new IPhone6sPlus();    ✓
2  sp.dial();    ✓
3  sp.facetime();    ×
4  sp.threeDTouch();    ×
```

Static type of *sp* is `SmartPhone`

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

Static Type and Polymorphism (1.2)

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone6sPlus extends IOS {
  void threeDTouch() { ... }
}
```

```
1  IOS ip = new IPhone6sPlus();    ✓
2  ip.dial();    ✓
3  ip.facetime();    ✓
4  ip.threeDTouch();    ×
```

Static type of *ip* is `IOS`

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

Static Type and Polymorphism (1.3)

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone6sPlus extends IOS {
  void threeDTouch() { ... }
}
```

```
1  IPhone6sPlus ip6sp = new IPhone6sPlus();   ✓
2  ip6sp.dial();   ✓
3  ip6sp.facetime();   ✓
4  ip6sp.threeDTouch();   ✓
```

Static type of *ip6sp* is `IPhone6sPlus`

⇒ can call all methods defined in `IPhone6sPlus` on *ip6sp*

Static Type and Polymorphism (1.4)

```
class SmartPhone {
  void dial() { ... }
}
class IOS extends SmartPhone {
  void facetime() { ... }
}
class IPhone6sPlus extends IOS {
  void threeDTouch() { ... }
}
```

```
1  SmartPhone sp = new IPhone6sPlus();    ✓
2  ((IPhone6sPlus) sp).dial();    ✓
3  ((IPhone6sPlus) sp).facetime();    ✓
4  ((IPhone6sPlus) sp).threeDTouch();    ✓
```

L4 is equivalent to the following two lines:

```
IPhone6sPlus ip6sp = (IPhone6sPlus) sp;
ip6sp.threeDTouch();
```

Static Type and 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 Call Arguments (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`:
  - `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*.

Polymorphism: Method Call Arguments (2.1)

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 Call Arguments (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 descendant of `s`' ***ST***.
  - *Valid* call ∵ `s`' temporary ***ST*** (`ResidentStudent`) is now a descendant class of `addRS`'s parameter `rs`' ***ST*** (`ResidentStudent`).
- But, there will be a `ClassCastException` at runtime!
  ∵ `s`' ***DT*** (`Student`) is **not** a descendant 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 Call Arguments (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 descendant of `s`' ***ST***.
  - *Valid* call ∵ `s`' temporary ***ST*** (`ResidentStudent`) is now a descendant class of `addRS`'s parameter `rs`' ***ST*** (`ResidentStudent`).
- But, there will be a *`ClassCastException`* at runtime!
  ∵ `s`' ***DT*** (`NonResidentStudent`) **not** descendant 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 Call Arguments (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 descendant of `s`' ***ST***.
  - *Valid* call ∵ `s`' temporary ***ST*** (`ResidentStudent`) is now a descendant class of `addRS`'s parameter `rs`' ***ST*** (`ResidentStudent`).
- And, there will be **no** *`ClassCastException`* at runtime!
  ∵ `s`' ***DT*** (`ResidentStudent`) is descendant 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 Call Arguments (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 descendant 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

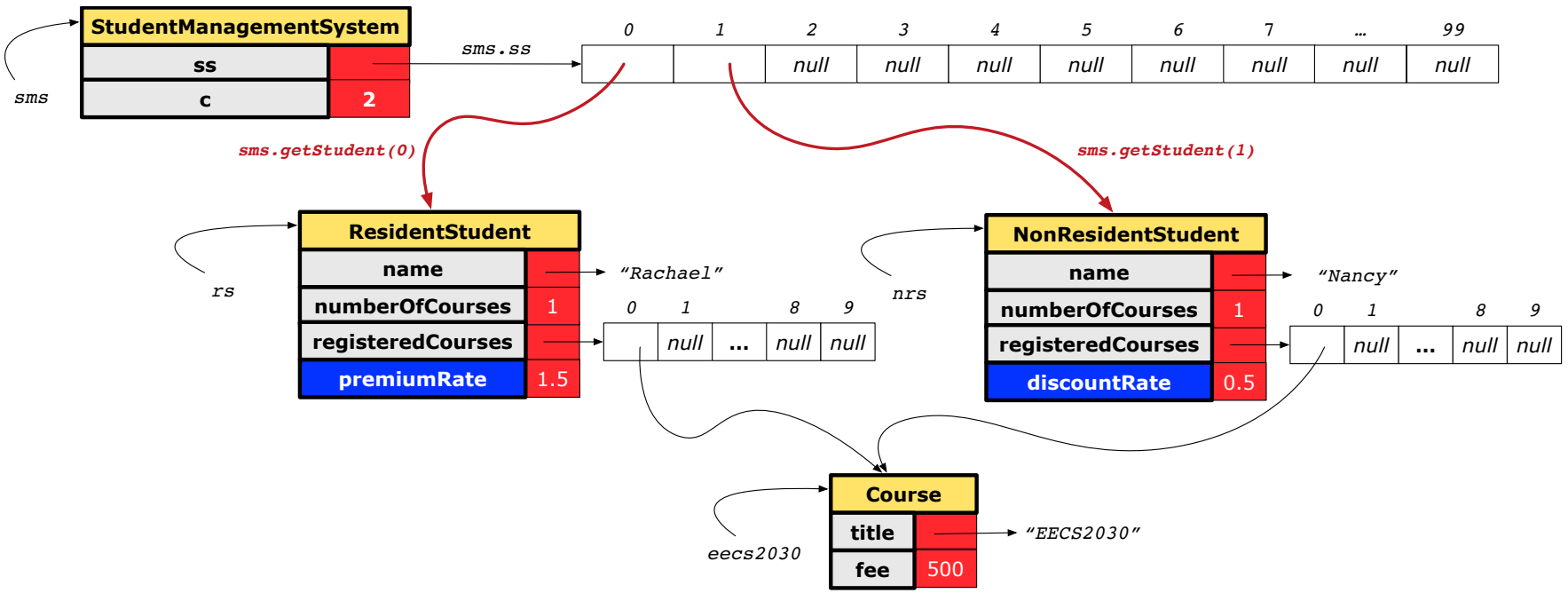
Polymorphism and Dynamic Binding: A Polymorphic Collection of Students (1)

```
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 Values (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 IllegalArgumentException("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`.

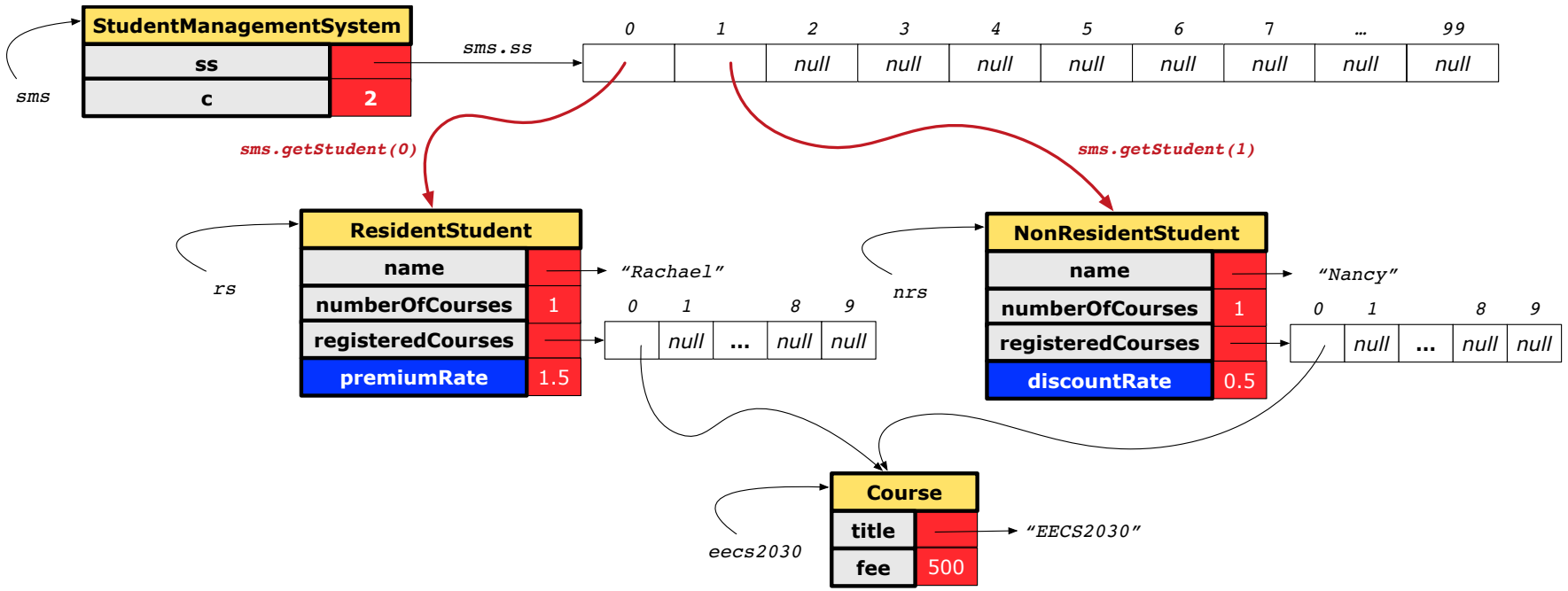
Polymorphism: Return Values (2)

```
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 Values (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); }
```

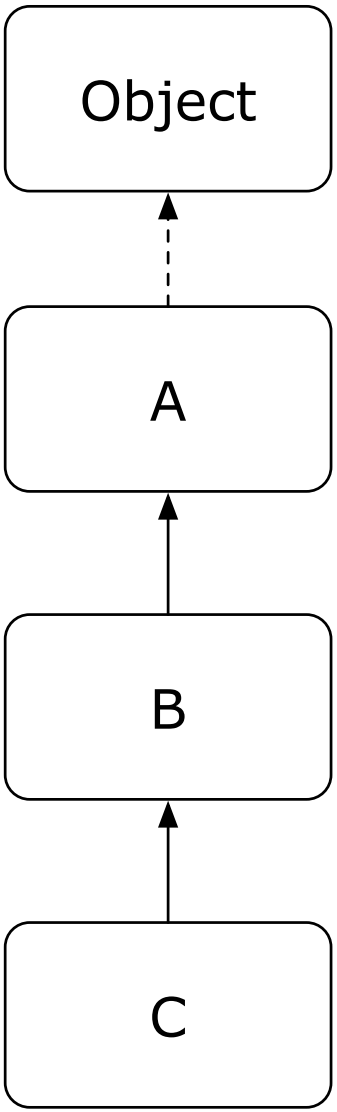
  - `String toString()`
    Returns a string representation of the object.
- Very often when you define new classes, you want to *redefine* / *override* the inherited definitions of `equals` and `toString`.

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)



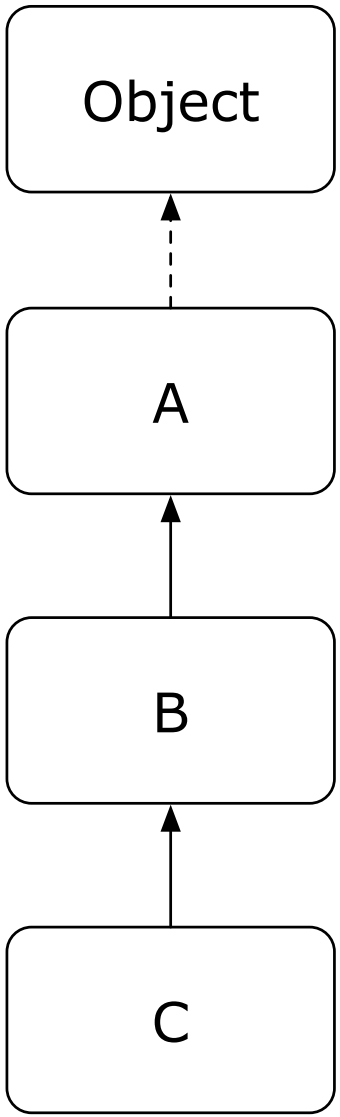

```
boolean equals (Object obj) {
  return this == obj;
}
```

```
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)




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

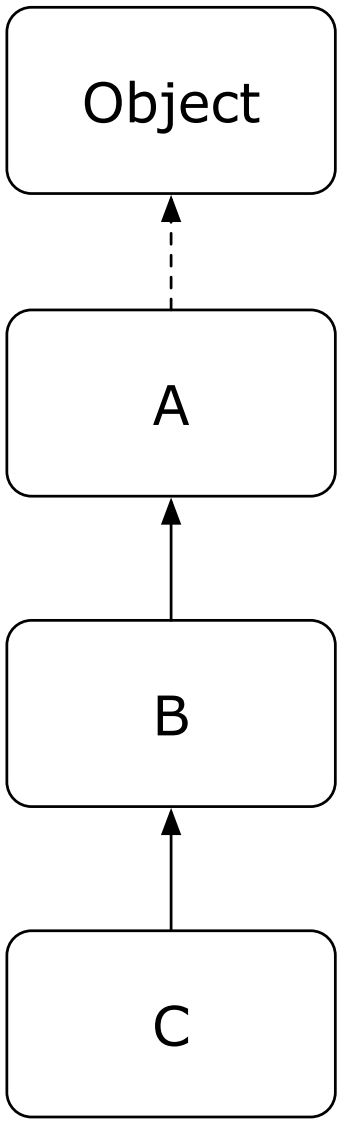
```
boolean equals (Object obj) {
  /* overridden version */
}
```

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

Overriding and Dynamic Binding (2.3)




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

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

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

Behaviour of Inherited `toString` Method (1)

```
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.

Behaviour of Inherited `toString` Method (2)

```
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)
```

Behaviour of Inherited `toString` Method (3)

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

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)

Index (7)

Abstract Classes and Interfaces

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

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)

```
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 *signature* only.
- ∴ `Polygon` cannot be used as a *dynamic type*
- Writing *new* `Polygon(...)` is forbidden!

Abstract Class (3)

```
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)

```
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)

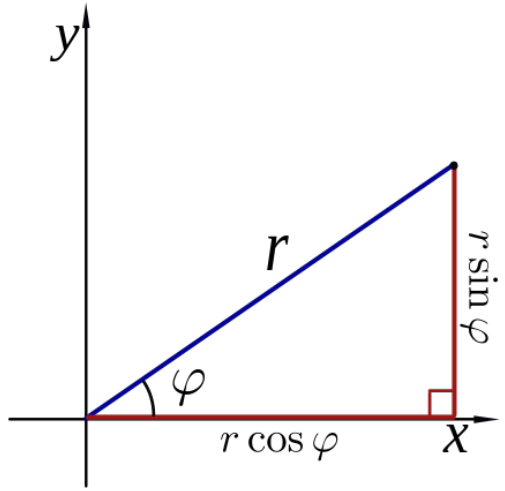
```
1    PolygonConstructor con = new PolygonConstructor();
2    double[] recSides = {3, 4, 3, 4};  p = con.getPolygon(recSides);
3    System.out.println(p instanceof Polygon);   ✓
4    System.out.println(p instanceof Rectangle);   ✓
5    System.out.println(p instanceof Triangle);   ×
6    System.out.println(p.getPerimeter()); /* 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)

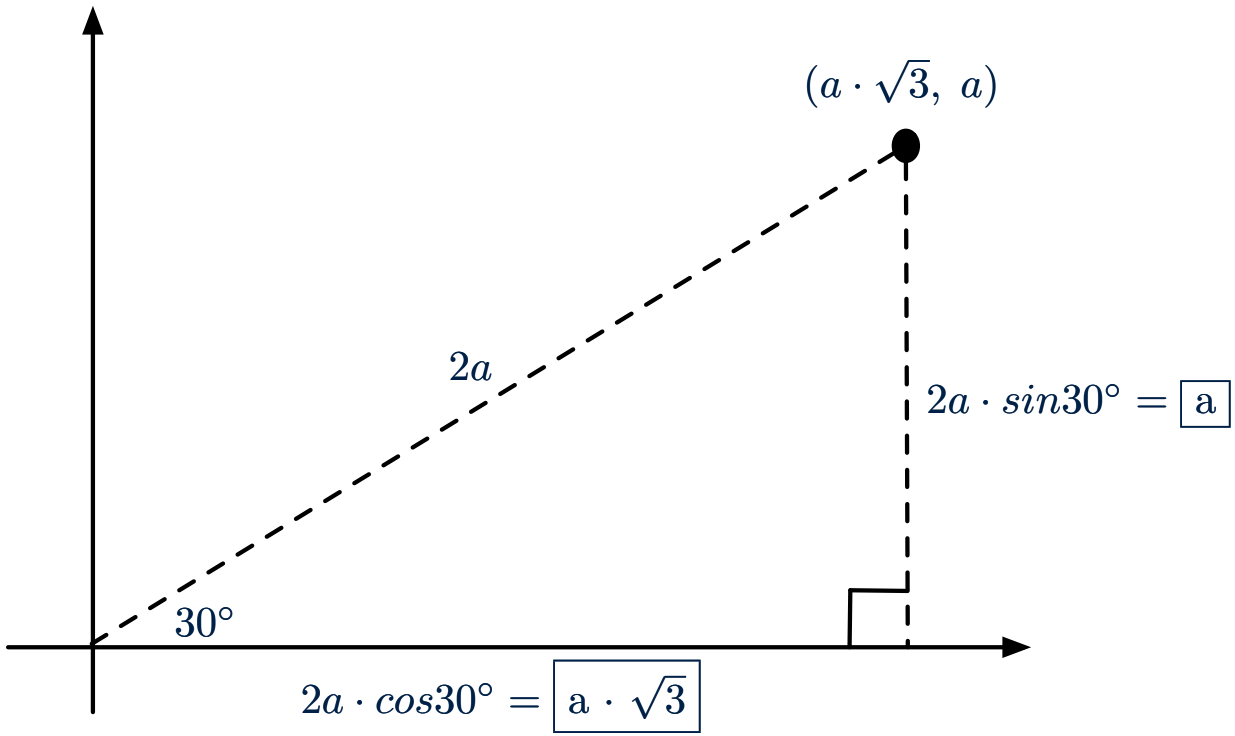
- We may implement `Point` using two representation systems:



  - 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 `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)

```
interface Point {
        double getX();
        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 *signatures* only.
- ∴ `Point` cannot be used as a *dynamic type*
- Writing *new* `Point(...)` is forbidden!

Interface (3)

```
public class CartesianPoint implements Point {
  double x;
  double y;
  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 {
  double phi;
  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, π/6)` allowed! [360° = 2π]

Interface (5)

```
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 signatures 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 signature of `getArea`, to be implemented by `Rectangle` and `Triangle`.

Index (1)

Index (2)

Recursion

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

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:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1 & \text{if } n \geq 1 \end{cases}$$

- 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.

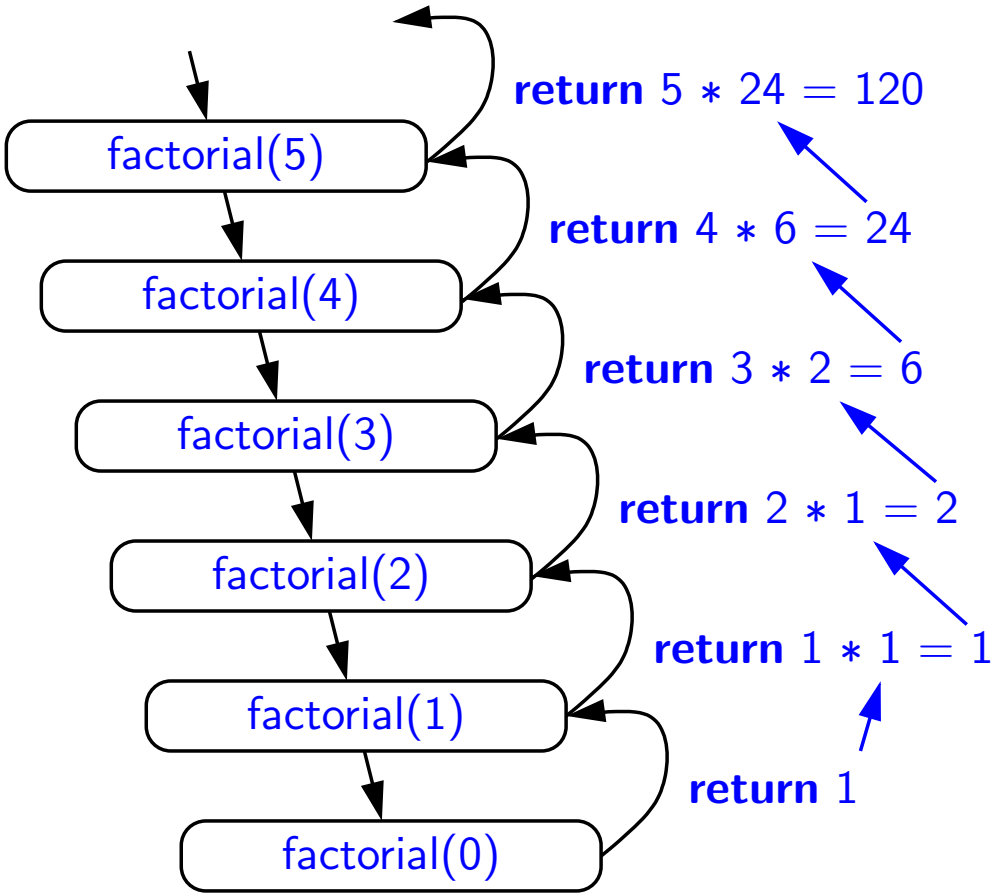
- Recursive Calls on Non-Smaller Problem Instances.

```
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 (2)



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 (1)

Recall the formal definition of calculating the n_{th} number in a Fibonacci series (denoted as F_n), which is already itself recursive:

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
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 (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*: ⟨⟩}

Java Library: String

```
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 $\longrightarrow$ Return *true* immediately.

Base Case 2: String of length 1 $\longrightarrow$ Return *true* immediately.

Recursive Case: String of length ≥ 2 $\longrightarrow$

- 1st and last characters match, **and**
- *the rest (i.e., middle) of the string* *is a palindrome* .

Recursion: Palindrome (2)

```
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 tail of string (i.e., all but the first character)

Return the concatenation of **1)** and **2)**.

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 $\geq 1 \longrightarrow$

1) Head of `s` (i.e., first character)

2) Number of occurrences of `c` in the tail of `s` (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)

```
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[0] = a[i - 1]; }
    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);
  }
}
```

Recursion: Is an Array Sorted? (1)

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 $\longrightarrow$ Return *true* immediately.

The base case is *true* $\because$ 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 $\longrightarrow$

- 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 positive* .

Recursion: Is an Array Sorted? (2)

```
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);
  }
}
```

Recursive Methods: Correctness Proofs

```
1  boolean allPositive(int[] a) { return allPosH(a, 0, a.length - 1); }
2  boolean allPosH(int[] a, int from, int to) {
3    if (from > to) { return true; }
4    else if(from == to) { return a[from] > 0; }
5    else { return a[from] > 0 && allPosH(a, from + 1, to); } }
```

- Via mathematical induction, prove that `allPosH` is correct:

  Base Cases

  - In an empty array, there is no non-positive number ∴ result is ***true***. **[L3]**
  - In an array of size 1, the only one elements determines the result. **[L4]**

  Inductive Cases

  - **Inductive Hypothesis**: `allPosH(a, from + 1, to)` returns ***true*** if a[from + 1], a[from + 2], . . . , a[to] are all positive; ***false*** otherwise.
  - `allPosH(a, from, to)` should return ***true*** if: **1)** a[from] is positive; and **2)** a[from + 1], a[from + 2], . . . , a[to] are all positive.
  - By *I.H.*, result is $a[from] > 0 \wedge$ `allPosH(a, from + 1, to)`. **[L5]**

- `allPositive(a)` is correct by invoking `allPosH(a, 0, a.length - 1)`, examining the entire array. **[L1]**

Recursion: Binary Search (1)

- **Searching Problem**

  Input: A number a and a *sorted* list of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

  Output: Whether or not a exists in the input list

- **An Efficient Recursive Solution**

  Base Case: Empty list $\longrightarrow$ *False*.

  Recursive Case: List of size $\geq 1 \longrightarrow$
  - *Compare* the *middle* element against a.
    - All elements to the left of *middle* are $\leq a$
    - All elements to the right of *middle* are $\geq a$
  - If the *middle* element *is* equal to $a \longrightarrow$ *True*.
  - If the *middle* element *is not* equal to a:
    - If $a <$ *middle*, recursively find a on the left half.
    - If $a >$ *middle*, recursively find a on the right half.

Recursion: Binary Search (2)

```
boolean binarySearch(int[] sorted, int key) {
  return binarySearchHelper(sorted, 0, sorted.length - 1, key);
}
boolean binarySearchHelper(int[] sorted, int from, int to, int key) {
  if (from > to) { /* base case 1: empty range */
    return false; }
  else if(from == to) { /* base case 2: range of one element */
    return sorted[from] == key; }
  else {
    int middle = (from + to) / 2;
    int middleValue = sorted[middle];
    if(key < middleValue) {
      return binarySearchHelper(sorted, from, middle - 1, key);
    }
    else if (key > middleValue) {
      return binarySearchHelper(sorted, middle + 1, to, key);
    }
    else  { return true; }
  }
}
```

Running Time: Binary Search (1)

We use *T(n)* to denote the running time function of a binary search, where *n* is the size of the input array.

$$\begin{cases} T(0) = 1 \\ T(1) = 1 \\ T(n) = T(\frac{n}{2}) + 1 \quad \texttt{where} \quad n \geq 2 \end{cases}$$

To solve this recurrence relation, we study the pattern of *T(n)* and observe how it reaches the *base case(s)*.

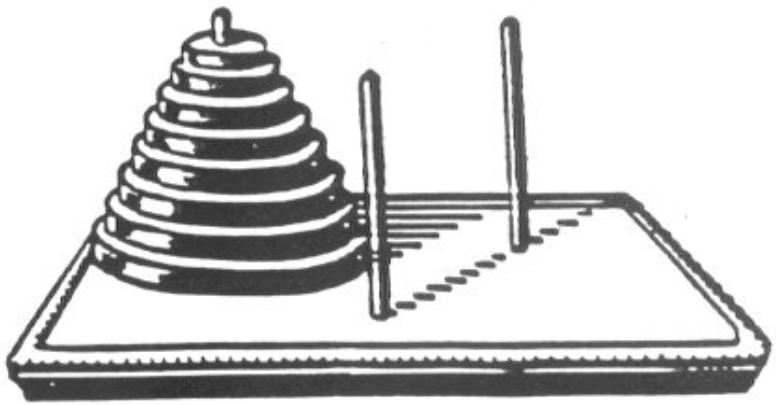
Running Time: Binary Search (2)

Without loss of generality, assume $n = 2^i$ for some non-negative i.

$$
\begin{aligned}
T(n) &= T(\tfrac{n}{2}) + 1 \\
&= (\underbrace{T(\tfrac{n}{4}) + 1}_{T(\frac{n}{2})}) + \underbrace{1}_{1 \text{ time}} \\
&= ((\underbrace{T(\tfrac{n}{8}) + 1}_{T(\frac{n}{4})}) + \underbrace{1) + 1}_{2 \text{ times}} \\
&= \ldots \\
&= (\,((\underbrace{1}_{T(\frac{n}{2^{\log n}})=T(1)}) + \underbrace{1) \ldots) + 1}_{\log n \text{ times}}
\end{aligned}
$$

$\therefore T(n)$ is $O(\log n)$

Tower of Hanoi: Specification



- *Given*: A tower of 8 disks, initially stacked in decreasing size on one of 3 pegs
- *Rules*:
  - Move only one disk at a time
  - Never move a larger disk onto a smaller one
- *Problem*: Transfer the entire tower to one of the other pegs.

Tower of Hanoi: A Recursive Solution

The general, recursive solution requires 3 steps:

1. Transfer the *n - 1* smallest disks to a different peg.
2. Move the largest to the remaining free peg.
3. Transfer the *n - 1* disks back onto the largest disk.

Tower of Hanoi in Java (1)

```
void towerOfHanoi(String[] disks) {
  tohHelper(disks, 0, disks.length - 1, 1, 3);
}
void tohHelper(String[] disks, int from, int to, int ori, int des){
  if(from > to) { }
  else if(from == to) {
    print("move " + disks[to] + " from " + ori + " to " + des);
  }
  else {
    int intermediate = 6 - ori - des;
    tohHelper(disks, from, to - 1, ori, intermediate);
    print("move " + disks[to] + " from " + ori + " to " + des);
    tohHelper(disks, from, to - 1, intermediate, des);
  }
}
```

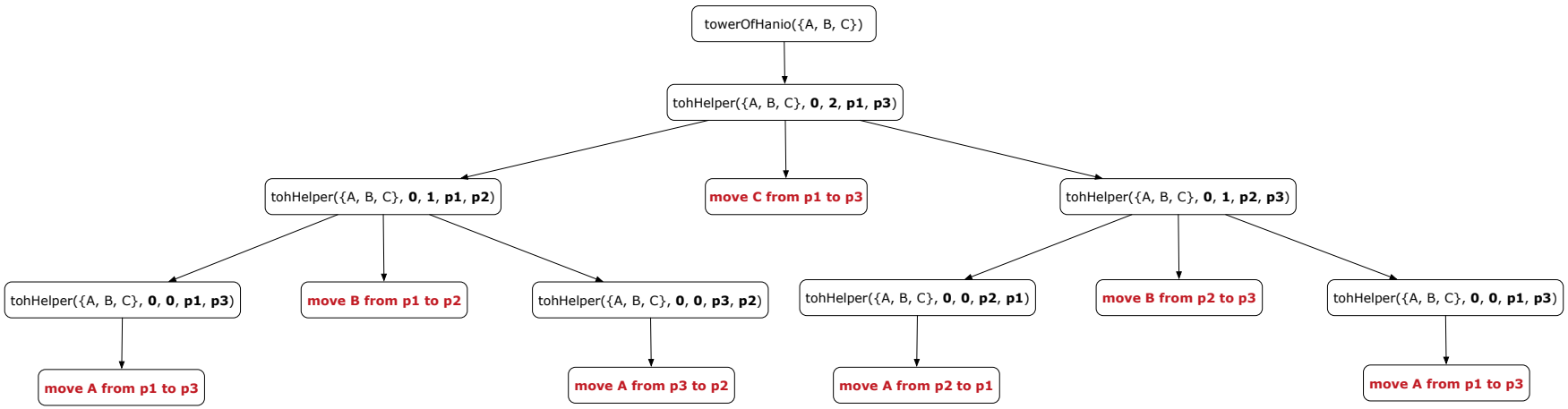
- `tohHelper(disks, from, to, ori, des)` moves disks $\{disks[from], disks[from+1], \ldots, disks[to]\}$ from peg *ori* to peg *des*.
- Peg id's are 1, 2, and 3 ⇒ The intermediate one is 6 – *ori* – *des*.

Tower of Hanoi in Java (2)

Say *ds* (disks) is $\{A, B, C\}$, where $A < B < C$.

$$
tohH(ds, \underbrace{0,2}_{\{A,B,C\}}, p1, p3) = \begin{cases}
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p1, p2) = \begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \{ \texttt{Move A: p1 to p3} \\
\texttt{Move B: p1 to p2} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p3, p2) = \{ \texttt{Move A: p3 to p2}
\end{cases} \\
\texttt{Move C: p1 to p3} \\
tohH(ds, \underbrace{0,1}_{\{A,B\}}, p2, p3) = \begin{cases}
tohH(ds, \underbrace{0,0}_{\{A\}}, p2, p1) = \{ \texttt{Move A: p2 to p1} \\
\texttt{Move B: p2 to p3} \\
tohH(ds, \underbrace{0,0}_{\{A\}}, p1, p3) = \{ \texttt{Move A: p1 to p3}
\end{cases}
\end{cases}
$$

Tower of Hanoi in Java (3)



Running Time: Tower of Hanoi (1)

- Generalize the problem by considering n disks.
- Let $T(n)$ denote the number of moves required to to transfer n disks from one to another under the rules.
- Recall the general solution pattern:
  1. Transfer the $n - 1$ smallest disks to a different peg.
  2. Move the largest to the remaining free peg.
  3. Transfer the $n - 1$ disks back onto the largest disk.
- We end up with the following recurrence relation that allows us to compute T_n for any n we like:

$$\begin{cases} T(1) = 1 \\ T(n) = 2 \times T(n-1) + 1 \quad \text{where } n > 0 \end{cases}$$

- To solve this recurrence relation, we study the pattern of T(n) and observe how it reaches the base case(s).

Running Time: Tower of Hanoi (2)

$$
\begin{aligned}
T(n) &= 2 \times T(n-1) + 1 \\
&= 2 \times (\underbrace{2 \times T(n-2) + 1}_{T(n-1)}) + 1 \\
&= 2 \times (2 \times (\underbrace{2 \times T(n-3) + 1}_{T(n-2)}) + 1) + 1 \\
&= \ldots \\
&= 2 \times (2 \times (2 \times (\underbrace{\cdots \times (\overbrace{2 \times T(1) + 1}^{T(2)}) + \ldots}_{T(n-3)}) + 1) + 1) + 1 \\
&= 2^{n-1} + (n-1)
\end{aligned}
$$

$\therefore T(n)$ is $O(2^n)$

Recursion: Merge Sort

- **Sorting Problem**

  Input: A list of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

  Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input list such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

- **Recursive Solution**

  Base Case 1: Empty list $\longrightarrow$ Automatically sorted.

  Base Case 2: List of size 1 $\longrightarrow$ Automatically sorted.

  Recursive Case: List of size $\geq 2 \longrightarrow$

  - Split the list into two (unsorted) halves: *L* and *R*;
  - **Recursively** sort *L* and *R*: *sortedL* and *sortedR*;
  - Return the *merge* of *sortedL* and *sortedR*.

Recursion: Merge Sort in Java (1)

```
/* Assumption:  L and R are both already sorted.  */
private List<Integer> merge(List<Integer> L, List<Integer> R) {
  List<Integer> merge = new ArrayList<>();
  if(L.isEmpty()||R.isEmpty()) { merge.addAll(L); merge.addAll(R); }
  else {
    int i = 0;
    int j = 0;
    while(i < L.size() && j < R.size()) {
      if( L.get(i) <= R.get(j) ) { merge.add(L.get(i)); i ++; }
      else { merge.add(R.get(j)); j ++; }
    }
    /* If i >= L.size(), then this for loop is skipped. */
    for(int k = i; k < L.size(); k ++) { merge.add(L.get(k)); }
    /* If j >= R.size(), then this for loop is skipped. */
    for(int k = j; k < R.size(); k ++) { merge.add(R.get(k)); }
  }
  return merge;
}
```

RT(merge)? [O(n)]

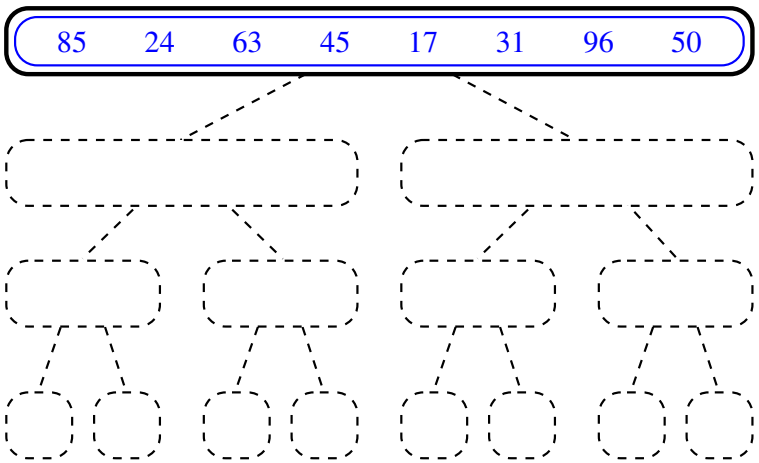
Recursion: Merge Sort in Java (2)

```
public List<Integer> sort (List<Integer> list) {
  List<Integer> sortedList;
  if(list.size() == 0) { sortedList = new ArrayList<>(); }
  else if(list.size() == 1) {
    sortedList = new ArrayList<>();
    sortedList.add(list.get(0));
  }
  else {
    int middle = list.size() / 2;
    List<Integer> left = list.subList(0, middle);
    List<Integer> right = list.subList(middle, list.size());
    List<Integer> sortedLeft = sort (left);
    List<Integer> sortedRight = sort (right);
    sortedList = merge (sortedLeft, sortedRight);
  }
  return sortedList;
}
```

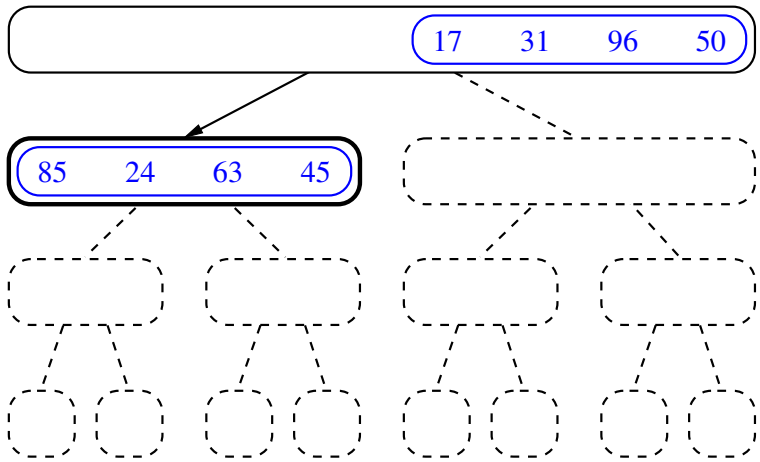
$$\text{RT(sort)} = \underbrace{RT(merge)}_{O(n)} \times \underbrace{\#\ \text{splits until size 0 or 1}}_{O(\log n)}$$

Recursion: Merge Sort Example (1)

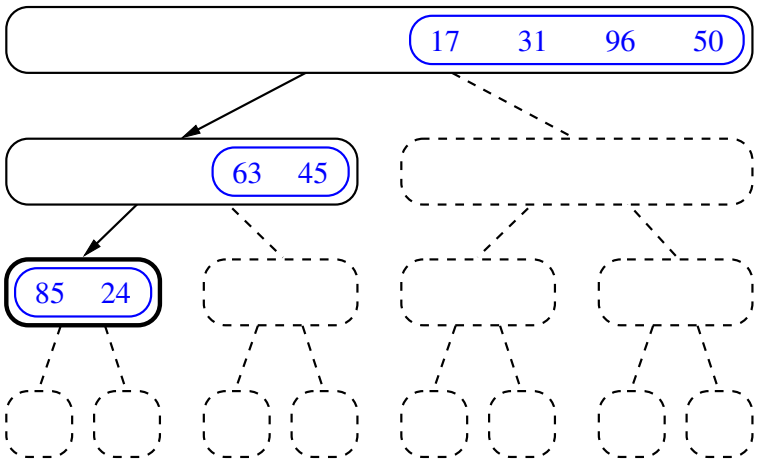
(1) Start with input list of size 8



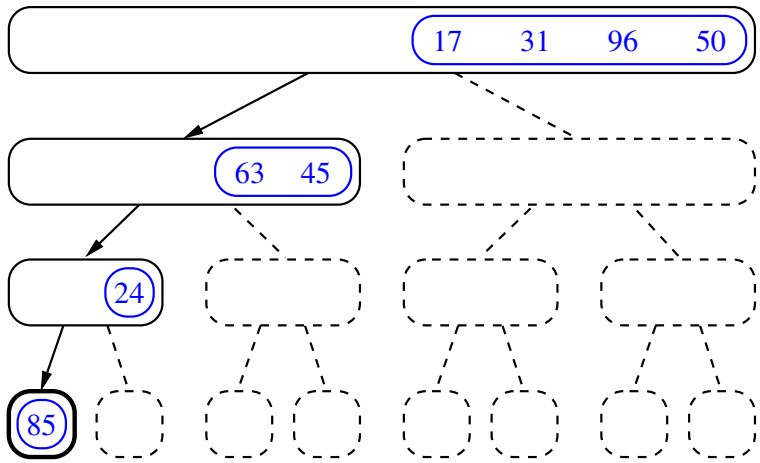
(2) Split and recur on L of size 4



(3) Split and recur on L of size 2

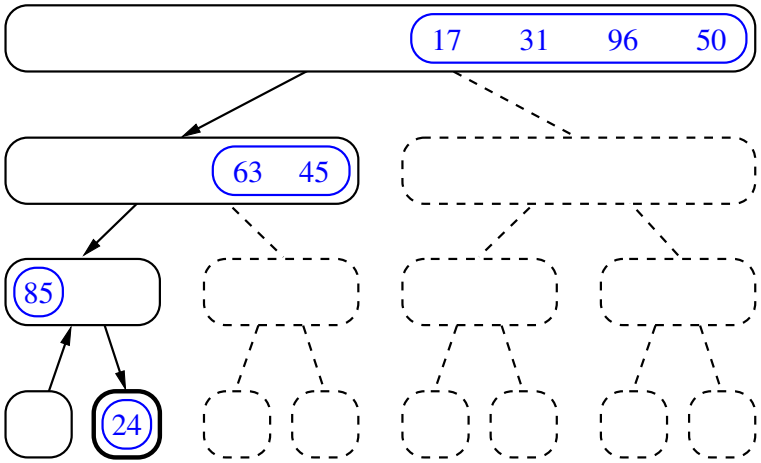


(4) Split and recur on L of size 1, *return*

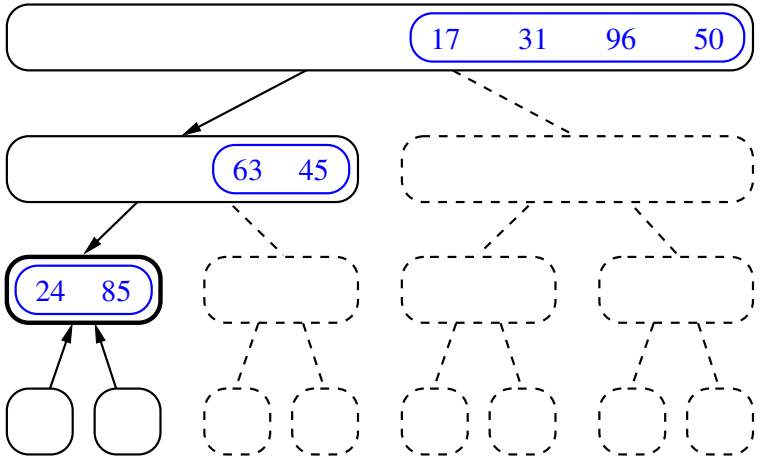


Recursion: Merge Sort Example (2)

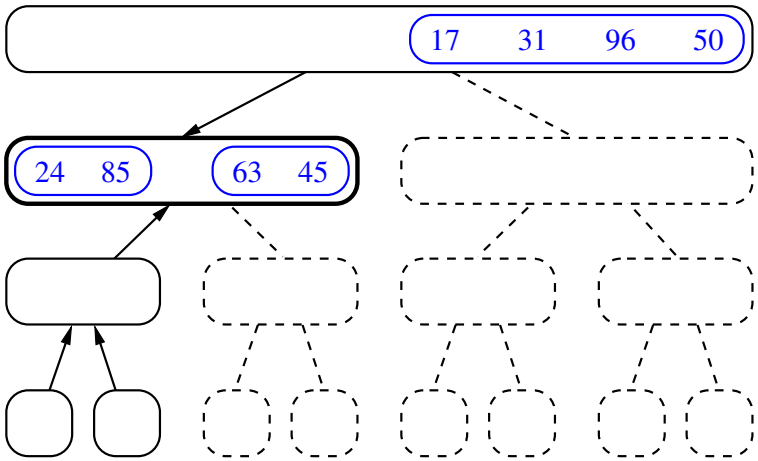
(5) Recur on R of size 1 and *return*



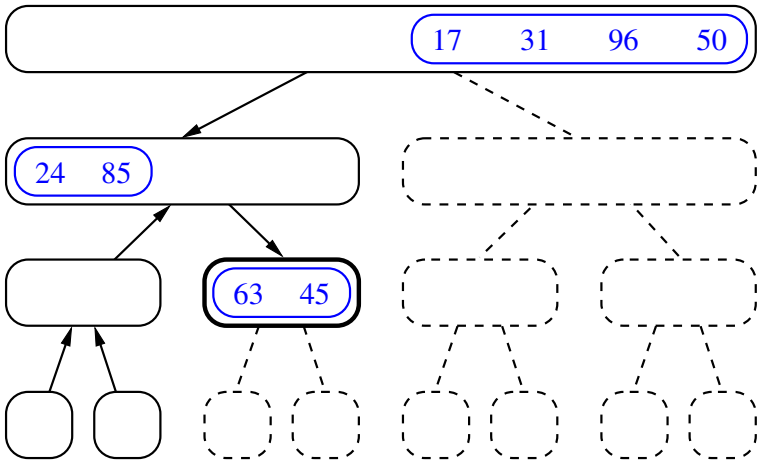
(6) Merge sorted L and R of sizes 1



(7) Return merged list of size 2

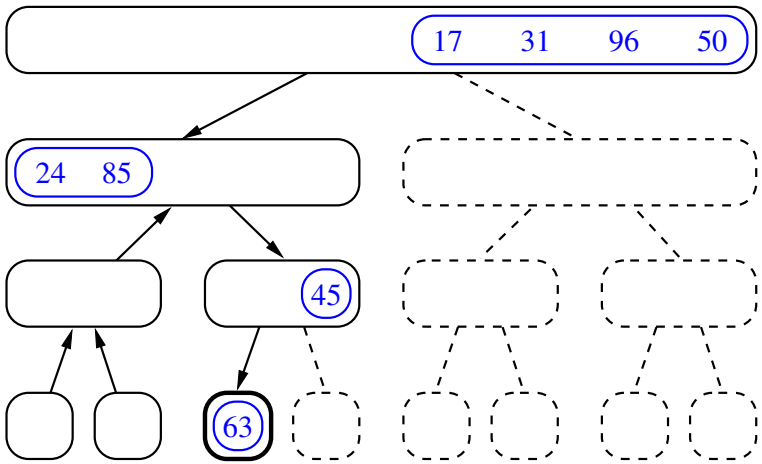


(8) Recur on R of size 2

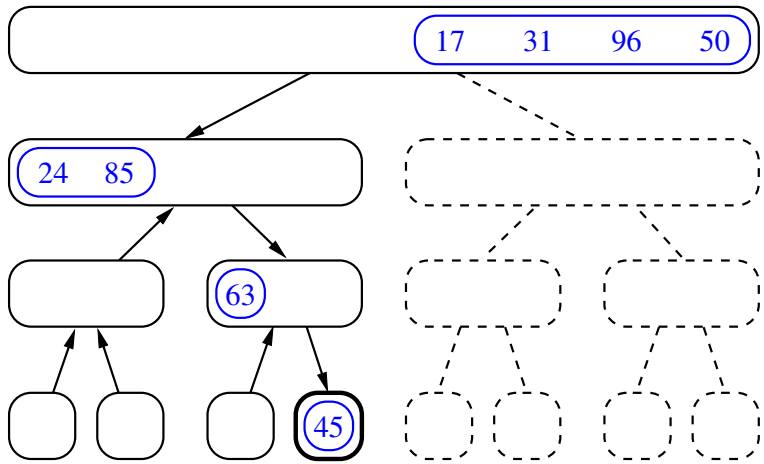


Recursion: Merge Sort Example (3)

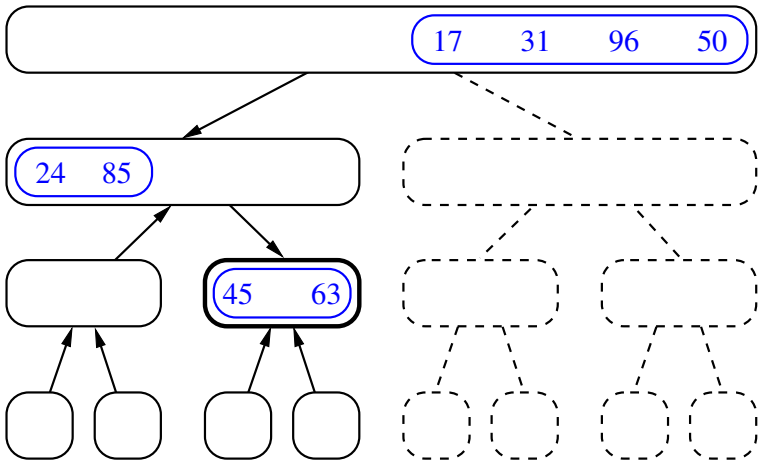
(9) Split and recur on L of size 1, *return*



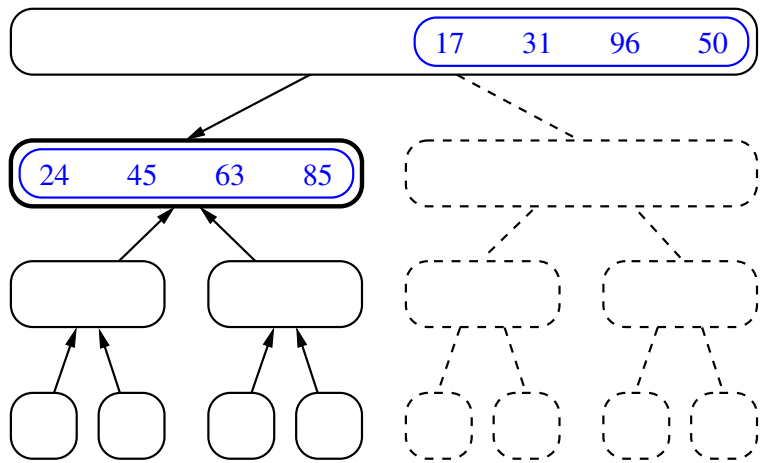
(10) Recur on R of size 1, *return*



(11) Merge sorted L and R of sizes 1, *return*

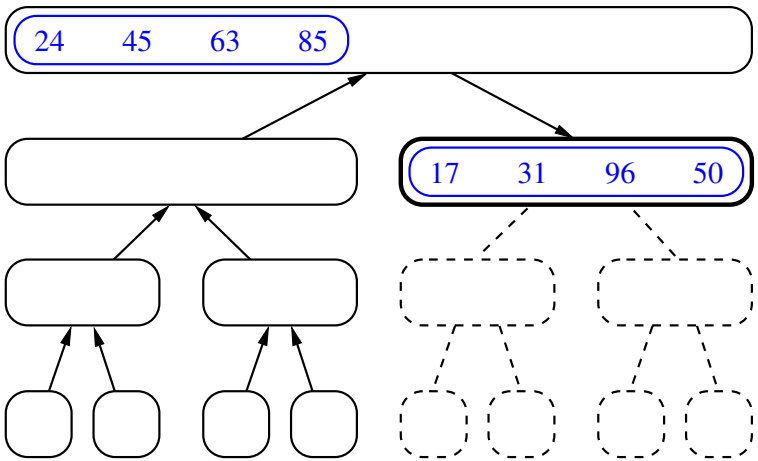


(12) Merge sorted L and R of sizes 2

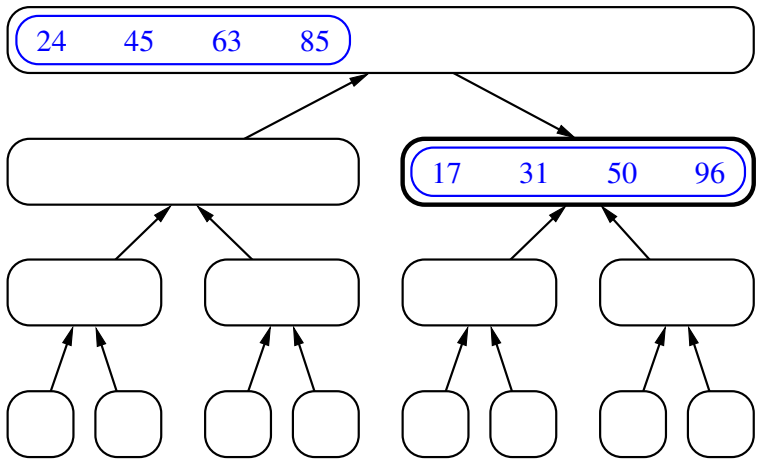


Recursion: Merge Sort Example (4)

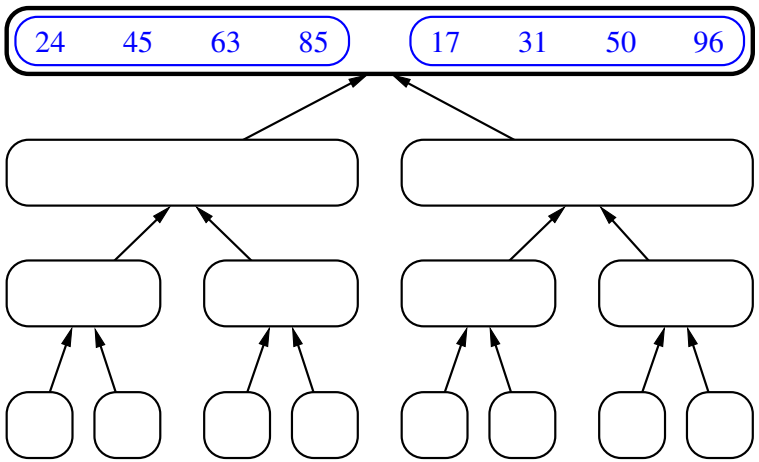
(13) Recur on R of size 4



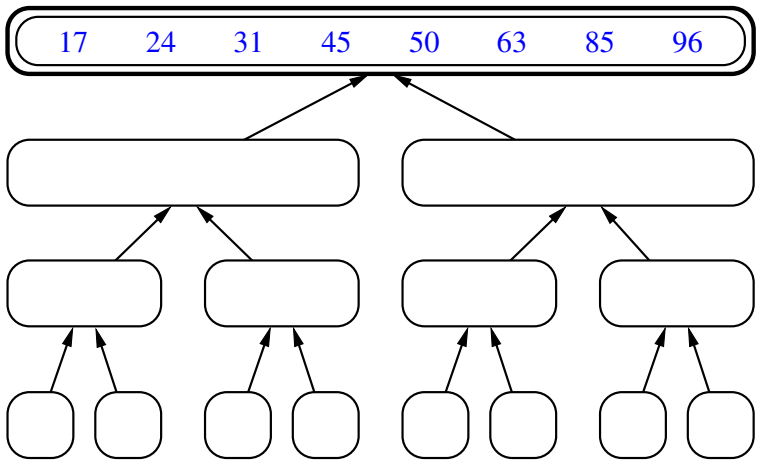
(14) *Return* a sorted list of size 4



(15) Merge sorted L and R of sizes 4

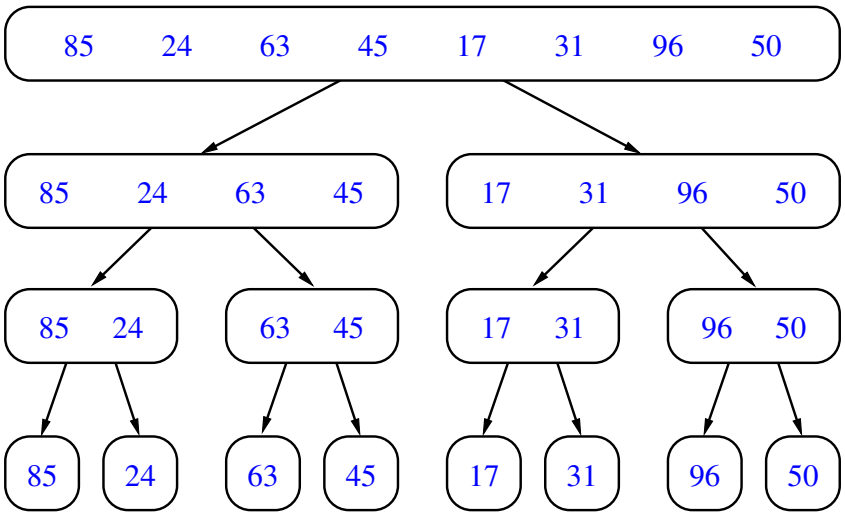


(16) *Return* a sorted list of size 8

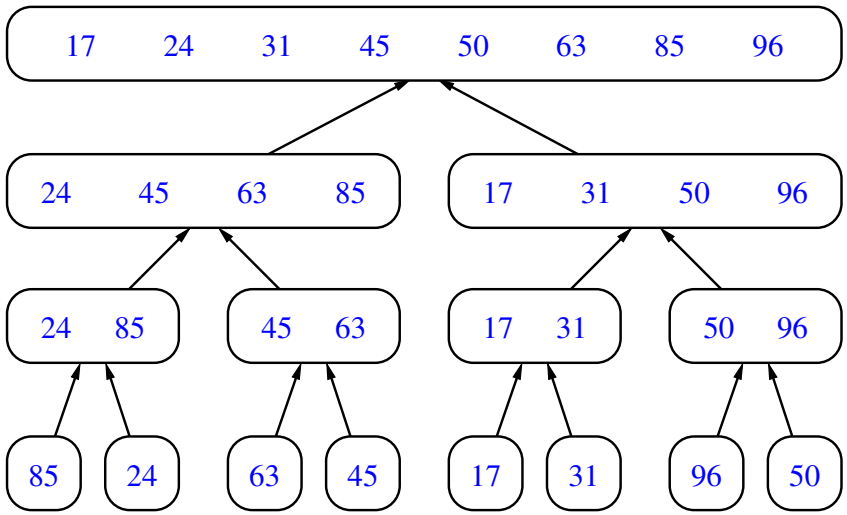


Recursion: Merge Sort Example (5)

(1) Recursion trees of *unsorted* lists



(2) Recursion trees of *sorted* lists



Recursion: Merge Sort Running Time (1)

Base Case 1: Empty list $\longrightarrow$ Automatically sorted. [O(1)]

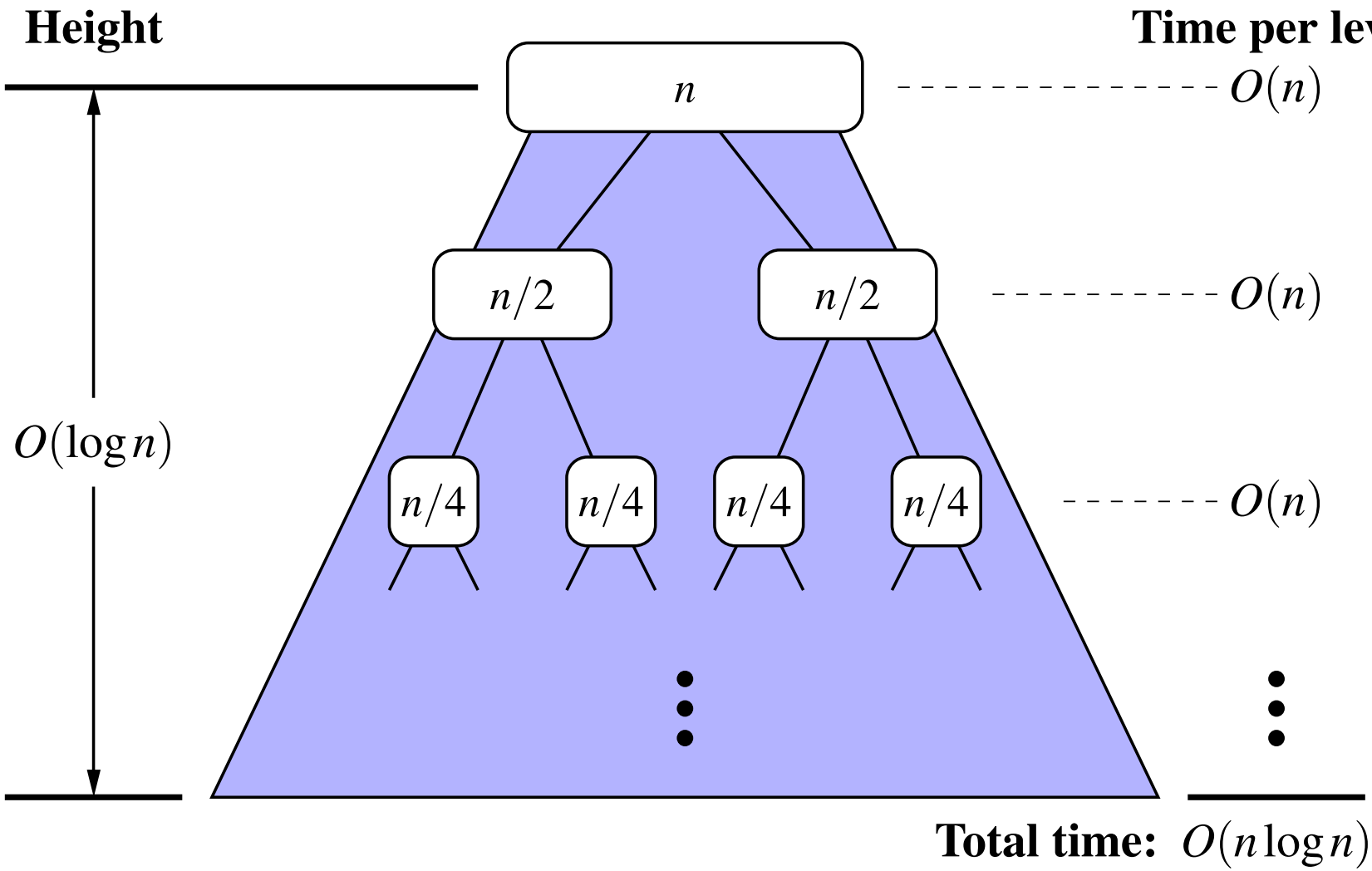
Base Case 2: List of size 1 $\longrightarrow$ Automatically sorted. [O(1)]

Recursive Case: List of size $\geq 2 \longrightarrow$

- Split the list into two (unsorted) halves: *L* and *R*; [O(1)]
- **Recursively** sort *L* and *R*: *sortedL* and *sortedR*;
  How many times to split until *L* and *R* have size 0 or 1? [*O(log n)*]
- Return the *merge* of *sortedL* and *sortedR*. [*O(n)*]

RT
= (**RT** each RC) × (# **RC**s)
= (**RT** merging *sortedL* and *sortedR*) × (# splits until bases)
= $n \cdot \log n$

Recursion: Merge Sort Running Time (2)



Beyond this lecture ...

- Notes on Recursion:

  `http://www.eecs.yorku.ca/~jackie/teaching/lectures/2017/F/EECS2030/slides/EECS2030_F17_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)

Index (2)

Index (3)

Index (4)

Generics in Java

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

Motivating Example: A Book of Objects

```
1  class Book {
2    String[] names;
3    Object[] records;
4    /* add a name-record pair to the book */
5    void add (String name, Object record) { ... }
6    /* return the record associated with a given name */
7    Object get (String name) { ... } }
```

Question: Which line has a type error?

```
1  Date birthday; String phoneNumber;
2  Book b; boolean isWednesday;
3  b = new Book();
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;
```

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 *descendant class* 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 *descendant classes* of `Object`.
  - So, **Line 5** and **Line 7** compile.

Motivating Example: Observations (2)

Due to *polymorphism*, the *dynamic types* of stored objects (e.g., `phoneNumber` and `birthday`) need not be the same.

- Methods supported in 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` *check* and *type cast* works.
- Can you see any potential problem(s)?
- **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.*)?

Motivating Example: Observations (2.2)

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?
  $\Rightarrow$ A *compilation error*
- When attempting to retrieve a record object from `b`, there is *no longer a need to check and cast*.
  $\because$ ***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., $+ : (\mathbb{Z} \times \mathbb{Z}) \rightarrow \mathbb{Z}$ 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 Generics: Design of a Generic Book

```
class Book <E> {
  String[] names;
  E [] records;
  /* add a name-record pair to the book */
  void add (String name, E record) { ... }
  /* return the record associated with a given name */
  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 not a `Date`]
  - **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.

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 *minimum* amount of features available for use.
  ∴ Exhaustive list of casts are unavoidable.

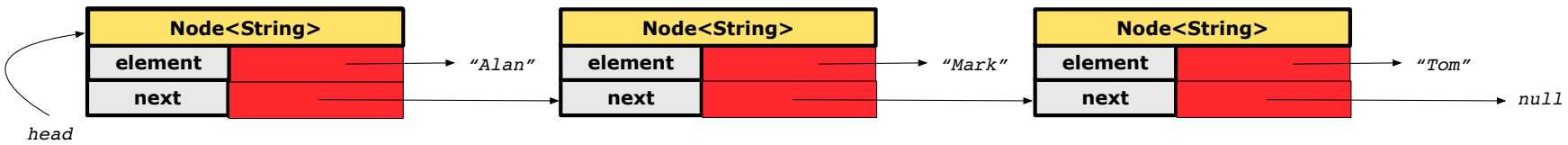
[***bad*** for extensibility and maintainability]

Generic Classes: Singly-Linked List (1)

```
public class Node< E > {
   private  E  element;
   private Node< E > next;
   public Node( E  e, Node< E > n) { element = e; next = n; }
   public  E  getElement() { return element; }
   public Node< E > getNext() { return next; }
   public void setNext(Node< E > n) { next = n; }
   public void setElement( E  e) { element = e; }
}
```

```
public class SinglyLinkedList< E > {
   private Node< E > head;
   private Node< E > tail;
   private int size = null;
   public void addFirst( E  e) { ... }
   Node< E > getNodeAt (int i) { ... }
   ...
}
```

Generic Classes: Singly-Linked List (2)



Approach 1

```
Node<String> tom = new Node<>("Tom", null);
Node<String> mark = new Node<>("Mark", tom);
Node<String> alan = new Node<>("Alan", mark);
```

Approach 2

```
Node<String> alan = new Node<>("Alan", null);
Node<String> mark = new Node<>("Mark", null);
Node<String> tom = new Node<>("Tom", null);
alan.setNext(mark);
mark.setNext(tom);
```

Generic Classes: Singly-Linked List (3)

Assume we are in the context of class `SinglyLinkedList`.

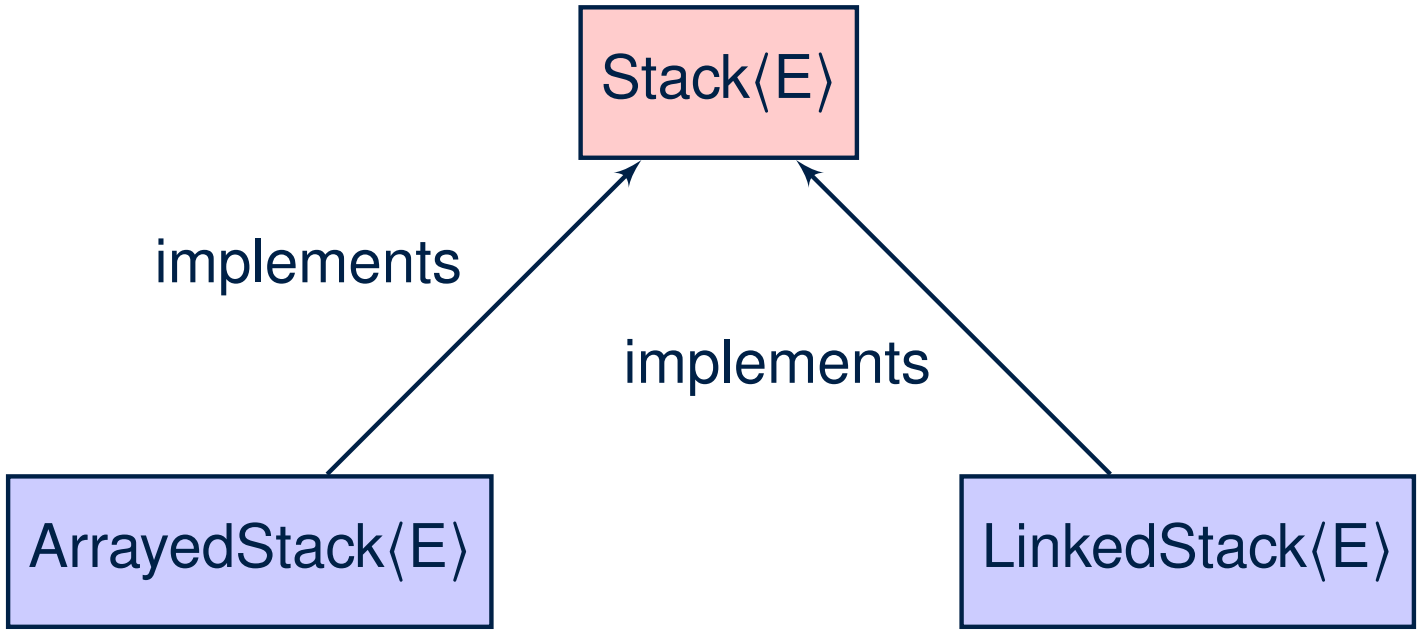
```
void addFirst (E e) {
   head = new Node<E>(e, head);
   if (size == 0) { tail = head; }
   size ++;
}
```

```
Node<E> getNodeAt (int i) {
   if (i < 0 || i >= size) {
     throw new IllegalArgumentException("Invalid Index"); }
   else {
      int index = 0;
      Node<E> current = head;
      while (index < i) {
         index ++; current = current.getNext();
      }
      return current;
   }
}
```

Generic Stack: Interface

```
public interface Stack< E > {
  public int size();
  public boolean isEmpty();
  public  E  top();
  public void push( E  e);
  public  E  pop();
}
```

Generic Stack: Architecture



Generic Stack: Array Implementation

```
public class ArrayedStack< E > implements Stack< E > {
  private static final int MAX_CAPACITY = 1000;
  private E[] data;
  private int t; /* top index */
  public ArrayedStack() {
    data = (E[]) new Object[MAX_CAPACITY];
    t = -1; }
  public int size() { return (t + 1); }
  public boolean isEmpty() { return (t == -1); }
  public E top() {
    if (isEmpty()) { /* Error: Empty Stack. */ }
    else { return data[t]; } }
  public void push(E e) {
    if (size() == MAX_CAPACITY) { /* Error: Stack Full. */ }
    else { t ++; data[t] = e; } }
  public E pop() {
    E result;
    if (isEmpty()) { /* Error: Empty Stack */ }
    else { result = data[t]; data[t] = null; t --; }
    return result; }
}
```

Generic Stack: SLL Implementation

```
public class LinkedStack< E > implements Stack< E > {
  private SinglyLinkedList< E > data;
  public LinkedStack() {
    data = new SinglyLinkedList< E >();
  }
  public int size() { return data.size(); }
  public boolean isEmpty() { return size() == 0; }
  public  E  top() {
    if (isEmpty()) { /* Error: Empty Stack. */ }
    else { return data.getFirst(); } }
  public void push( E  e) {
    data.addFirst(e); }
  public  E  pop() {
    E  result;
    if (isEmpty()) { /* Error: Empty Stack */ }
    else { result = top(); data.removeFirst(); }
    return result; }
}
```

Generic Stack: Testing Both Implementations

```
@Test
public void testPolymorphicStacks() {
  Stack<String> s = new ArrayedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());

  s = new LinkedStack<>();
  s.push("Alan"); /* dynamic binding */
  s.push("Mark"); /* dynamic binding */
  s.push("Tom"); /* dynamic binding */
  assertTrue(s.size() == 3 && !s.isEmpty());
  assertEquals("Tom", s.top());
}
```

Beyond this lecture . . .

- Study `https://docs.oracle.com/javase/tutorial/java/generics/index.html` for further details on Java generics.

Index (1)

Index (2)

Wrap-Up

EECS2030 B: Advanced
Object Oriented Programming
Fall 2018

CHEN-WEI WANG

What You Learned (1)

- *Procedural Programming in Java*
  - Exceptions
  - Recursion (implementation, running time, correctness)
- *Data Structures*
  - Arrays
  - Maps and Hash Tables

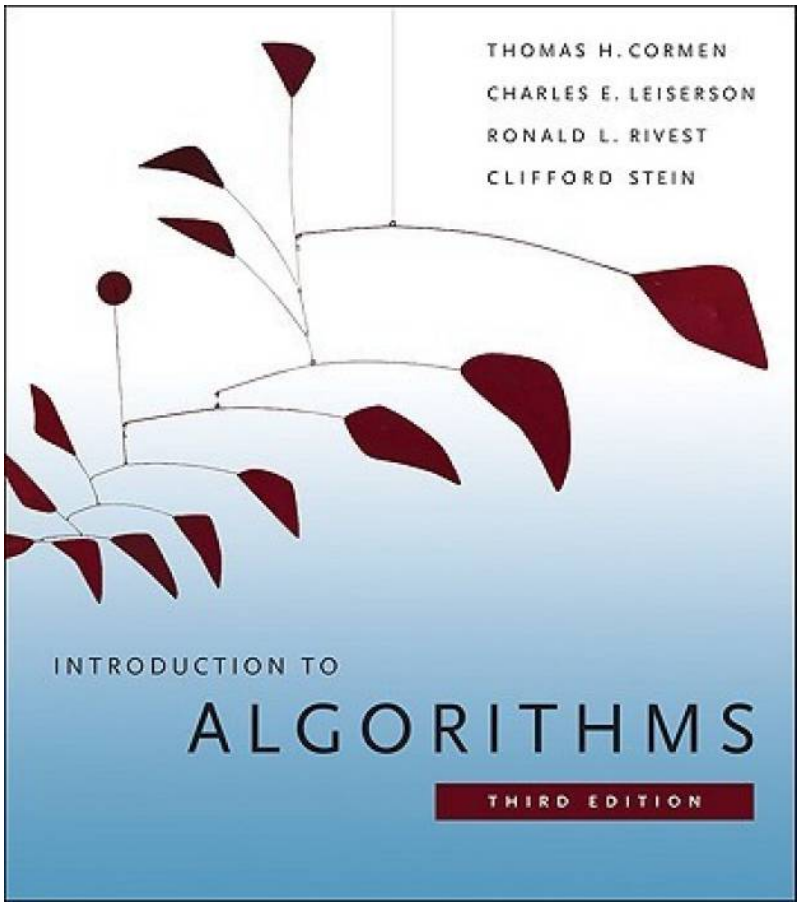
What You Learned (2)

- *Object-Oriented Programming in Java*
  - classes, attributes, encapsulation, objects, reference data types
  - methods: constructors, accessors, mutators, helper
  - dot notation, context objects
  - aliasing
  - inheritance:
    - code reuse
    - expectations
    - static vs. dynamic types
    - rules of substitutions
    - casts and `instanceof` checks
    - polymorphism and method arguments/return values
    - method overriding and dynamic binding: e.g., `equals`
    - abstract classes vs. interfaces
    - generics (vs. collection of `Object`) [Optional]

What You Learned (3)

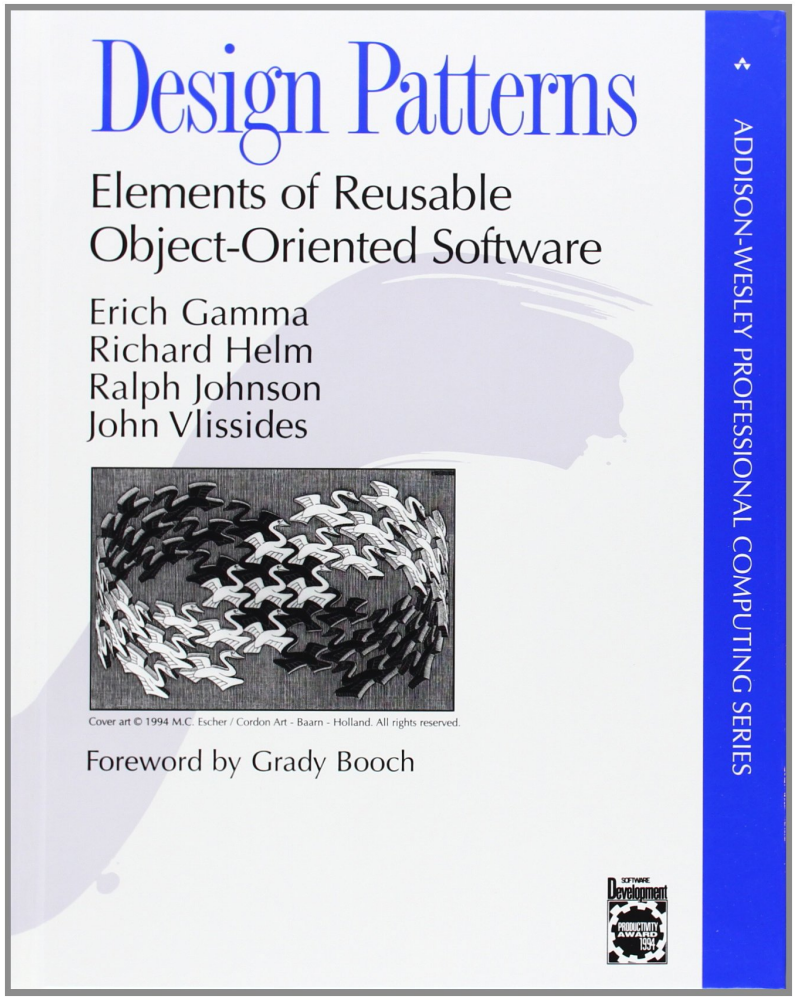
- *Integrated Development Environment (IDE) for Java: Eclipse*
  - Break Point and Debugger
  - Unit Testing using JUnit

Beyond this course. . . (1)



- *Introduction to Algorithms (3rd Ed.)* by Cormen, *etc.*
- DS by DS, Algo. by Algo.:
  - *Understand* math analysis
  - *Read* pseudo code
  - *Translate* into Java code
  - *Write and pass* JUnit tests

Beyond this course. . . (2)



- *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, *etc.*
- Pattern by Pattern:
  - *Understand* the problem
  - *Read* the solution (not in Java)
  - *Translate* into Java code
  - *Write and pass* JUnit tests

Beyond this course. . . (3)

Visit my lectures on *EECS3311 Software Design*:

`http://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS3311_F18`

- Design by Contracts
- Design Patterns
- Program Verification

Wish You All the Best

- What you have learned will be **assumed** in `EECS2011`.
- Logic is your friend: Learn/Review `EECS1019/EECS1090`.
- Do **not** abandon Java during the break!!
- Feel free to get in touch and let me know how you're doing :D