FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

– Chapter 2: Java Foundations II

Outline

- Object-oriented programming
- Class, object, reference, constructor, and method overloading
- The `this` and `static` keywords
- Package and import
- Access control modifier
- Inheritance, method overriding, and the `final` and `super` keyword
- Polymorphism, object casting
- Abstract class, interface
- Exception
- Nested class
- Iterator

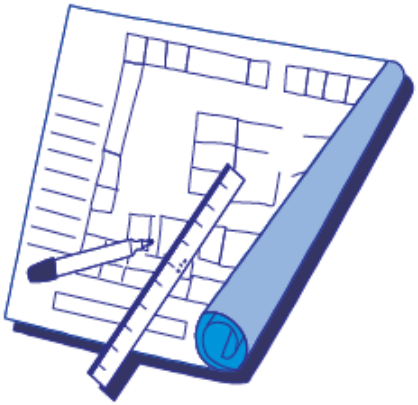
Goal of Object Oriented Programming

- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment
- Reusability
  - The same code should be usable as a component of different systems in various applications

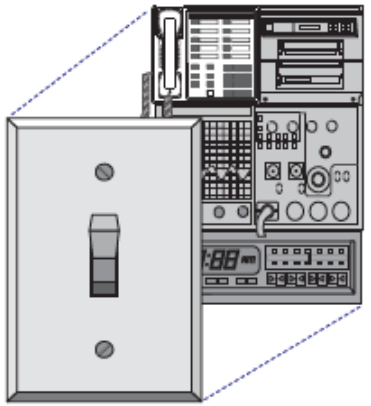
Object-Oriented Design Principles

- Modularity
- Abstraction
- Encapsulation

Modularity



Abstraction



Encapsulation

Design Patterns

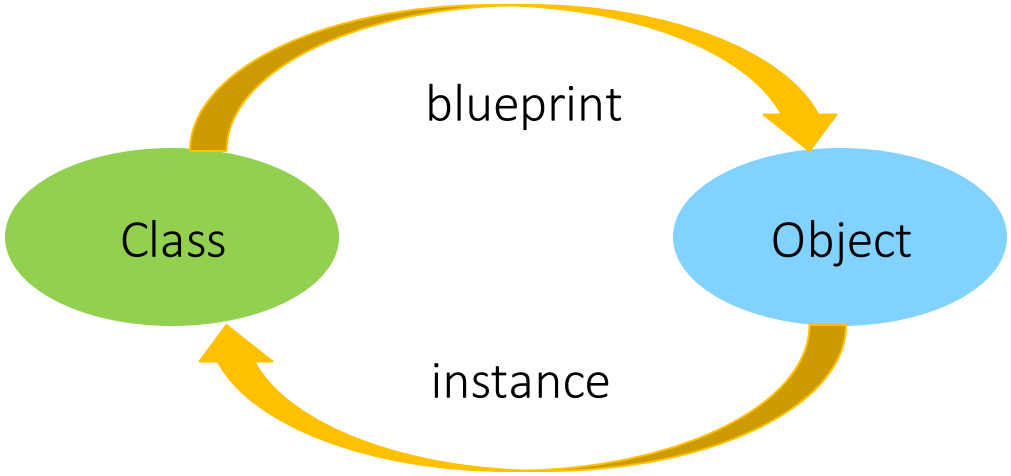
- Algorithmic patterns:
  - Recursion
  - Amortization
  - Divide-and-conquer
  - Prune-and-search
  - Brute force
  - Dynamic programming
  - The greedy method
- Software design patterns:
  - Iterator
  - Adapter
  - Position
  - Composition
  - Template method
  - Locator
  - Factory method

Object-Oriented Software Design

- Responsibilities: Divide the work into different actors, each with a different responsibility
- Independence: Define the work for each class to be as independent from other classes as possible
- Behaviors: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it

Classes and Objects

- Each object created in a program is an instance of a class
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects



Class Definitions

- The class definition typically specifies instance variables, also known as data members, attributes, or fields, that the object contains, as well as the methods, also known as member functions, that the object can execute. The critical members of a class in Java are
  - **Instance variables**, represent the data associated with an object of a class. Instance variables must have a type, which can either be a base type (such as `int`, `float`, or `double`) or any class type
  - **Methods** in Java are blocks of code that can be called to perform actions. Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an accessor method, while an update method is one that may change one or more instance variables when called

Class Definitions (cont'd.)

- Instance variable

  modifier type name = values (or default values)

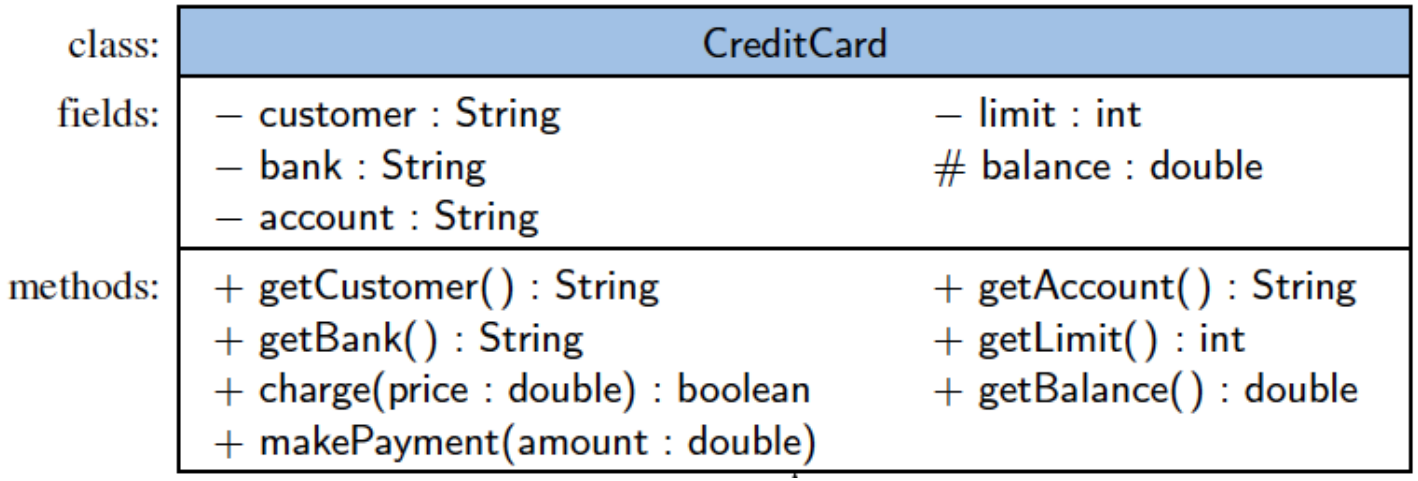
  - Example: `private int x = 20`
- Methods
  - Local variables

```
modifier modifier returnType name (arg, arg) {
        statements
}
```

  - Examples: `public static void main()`

Unified Modeling Language (UML)

- A class diagram has three portions
  - The name of the class
  - The recommended instance variables
  - The recommended methods of the class
- Example:



Creating and Using Objects

- Classes are known as **reference types** in Java, and a variable of that type is known as a **reference variable**
- A reference variable is capable of storing the location (i.e., memory address) of an object from the declared class
  - So we might assign it to reference an existing instance or a newly constructed instance
  - A reference variable can also store a special value, `null`, that represents the lack of an object

Parameters

- A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method
  - A parameter consists of two parts, the parameter type and the parameter name
  - If a method has no parameters, then only an empty pair of parentheses is used
- All parameters in Java are **passed by value**, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body
  - So if we pass an `int` variable to a method, then that variable's integer value is copied
  - The method can change the copy but not the original
  - If we pass an object reference as a parameter to a method, then the reference is copied as well

The Dot Operator

- One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class
- This access is performed with the dot (“.”) operator
- We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters

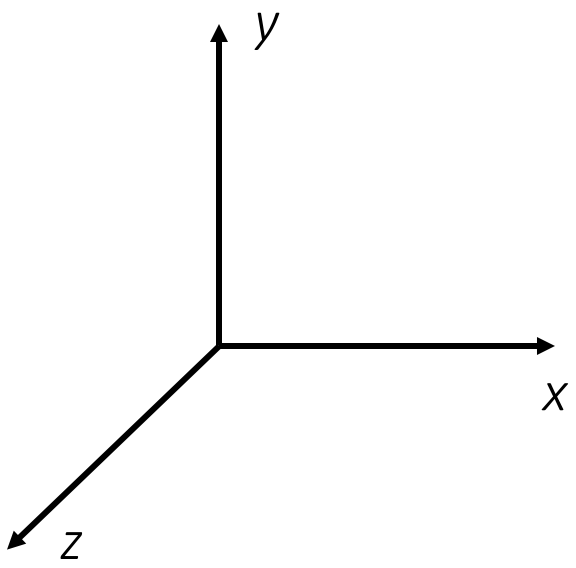
Constructors

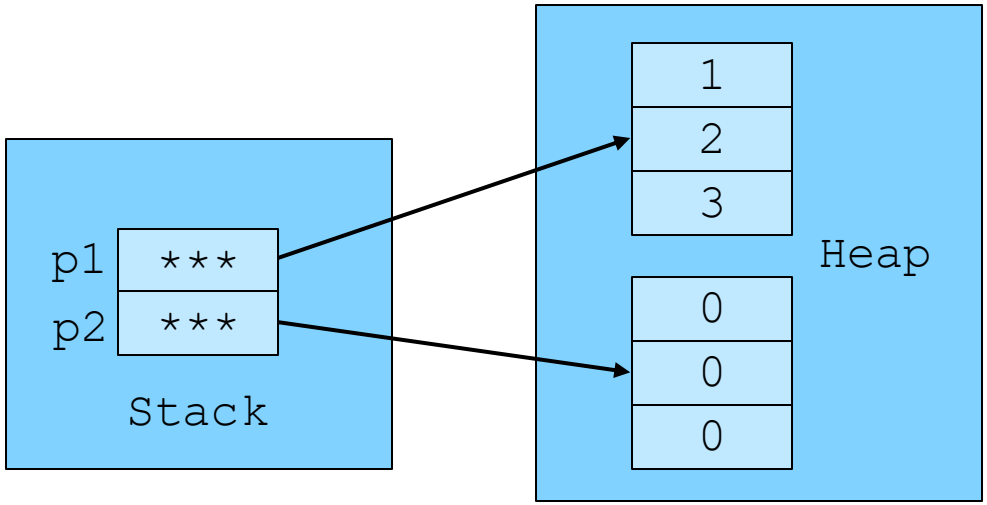
- A user can create an instance of a class by using the **new** operator with a method that has the same name as the class
- Such a method, known as a **constructor**, has as its responsibility is to establish the state of a newly object with appropriate initial values for its instance variables

Data type	Default value
`boolean`	`false`
`char`	`\u0000`
`int,short,byte / long`	`0 / 0L`
`float /double`	`0.0f / 0.0d`
`any reference type`	`null`

Exercise 2.1

- Create a class `Point` with coordinate (x, y, z)
- Calculate the distance between two points `p1` and `p2` by using the class `Point.java`





Point.java

TestPoint.java

Method Overloading

- Method overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different
- Argument lists could differ in:
  - Number of parameters
  - Data type of parameters
  - Sequence of Data type of parameters
- Constructor overloading that allows a class to have more than one constructors having different argument lists

- Find errors and fix them

```
    void max(float a, float b){
       System.out.println((a>b)? a : b);}

    void max(short a, short b){
       System.out.println((a>b)? a : b);}

    void max(int a, int b, int c){
       System.out.println(((a+b)>c)?(a+b):c);}

    int max(int a, int b){
       System.out.println((a>b)? a : b);}

    public static void main(String[] args){
       TestOverload to = new TestOverload();
       to.max(3,4);
       to.max(3,4,5);
       to.max(1.2f, 2.3f);
    }
}
```

TestOverload.java

Signatures

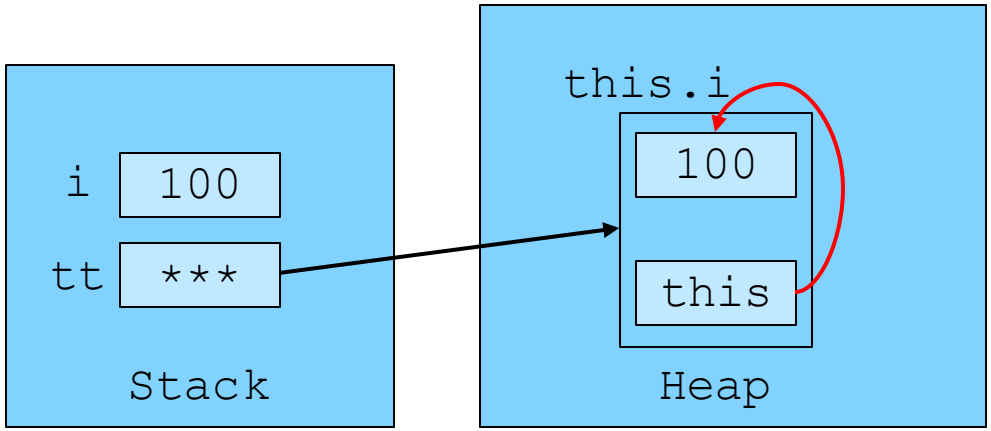
- If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types
- A method's name combined with the number and types of its parameters is called a method's **signature**, for it takes all of these parts to determine the actual method to perform for a certain method call
- A reference variable v can be viewed as a "pointer" to some object o

The Keyword `this`

- Within the body of a method in Java, the keyword this is automatically defined as a reference to the instance upon which the method was invoked. There are three common uses:
  - To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument
  - To differentiate between an instance variable and a local variable with the same name
  - To allow one constructor body to invoke another constructor body

Example

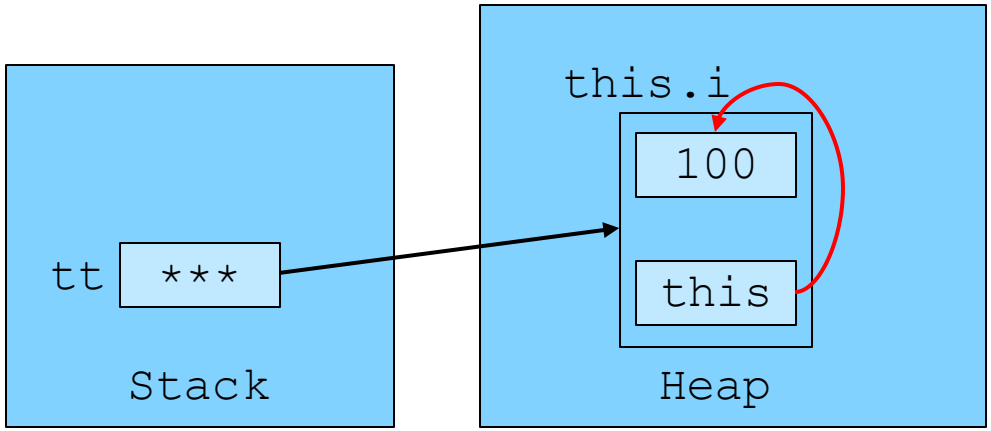
```
public class TestThis {
    int i = 0;
    TestThis (int i){
        this.i = i;}

    TestThis increment(){
        i++;
        return this;}

    void print(){
        System.out.println("i = " + i);}

    public static void main(String args[]){
        TestThis tt = new TestThis(100);
        tt.increment().increment().print();}
}
```



TestThis.java

Example (cont'd.)

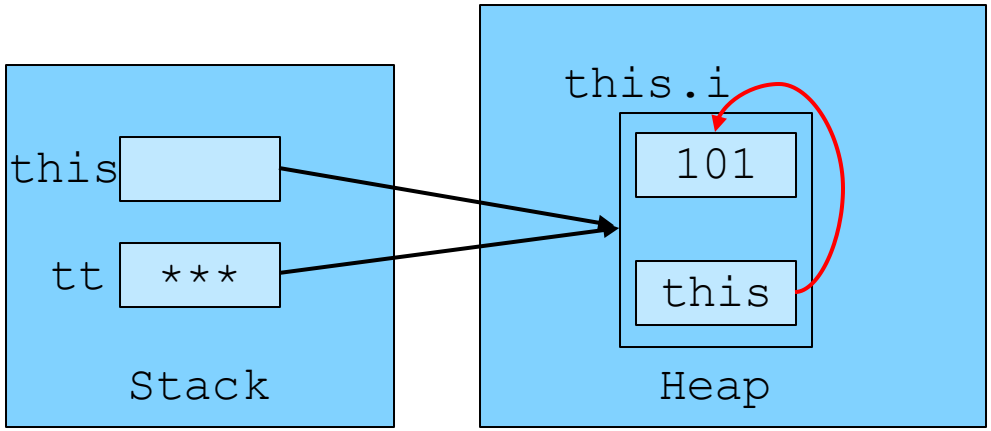
```
public class TestThis {
    int i = 0;
    TestThis (int i){
        this.i = i;}

    TestThis increment(){
        i++;
        return this;}

    void print(){
        System.out.println("i = " + i);}

    public static void main(String args[]){
        TestThis tt = new TestThis(100);
        tt.increment().increment().print();}
}
```



TestThis.java

Example (cont'd.)

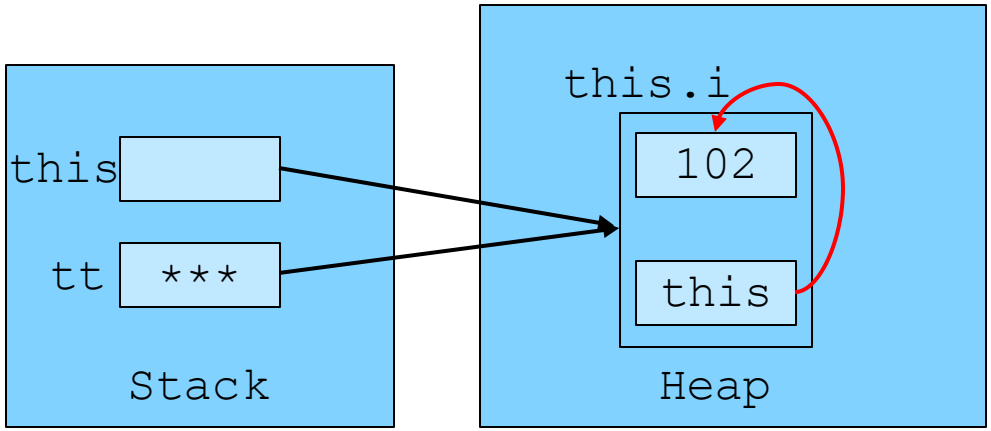
```
public class TestThis {
    int i = 0;
    TestThis (int i){
        this.i = i;}

    TestThis increment(){
        i++;
        return this;}

    void print(){
        System.out.println("i = " + i);}

    public static void main(String args[]){
        TestThis tt = new TestThis(100);
        tt.increment().increment().print();}
}
```



TestThis.java

Example (cont'd.)

```
public class TestThis {
    int i = 0;
    TestThis (int i){
        this.i = i;}

    TestThis increment(){
        i++;
        return this;}

    void print(){
        System.out.println("i = " + i);}

    public static void main(String args[]){
        TestThis tt = new TestThis(100);
        tt.increment().increment().print();}
}
```



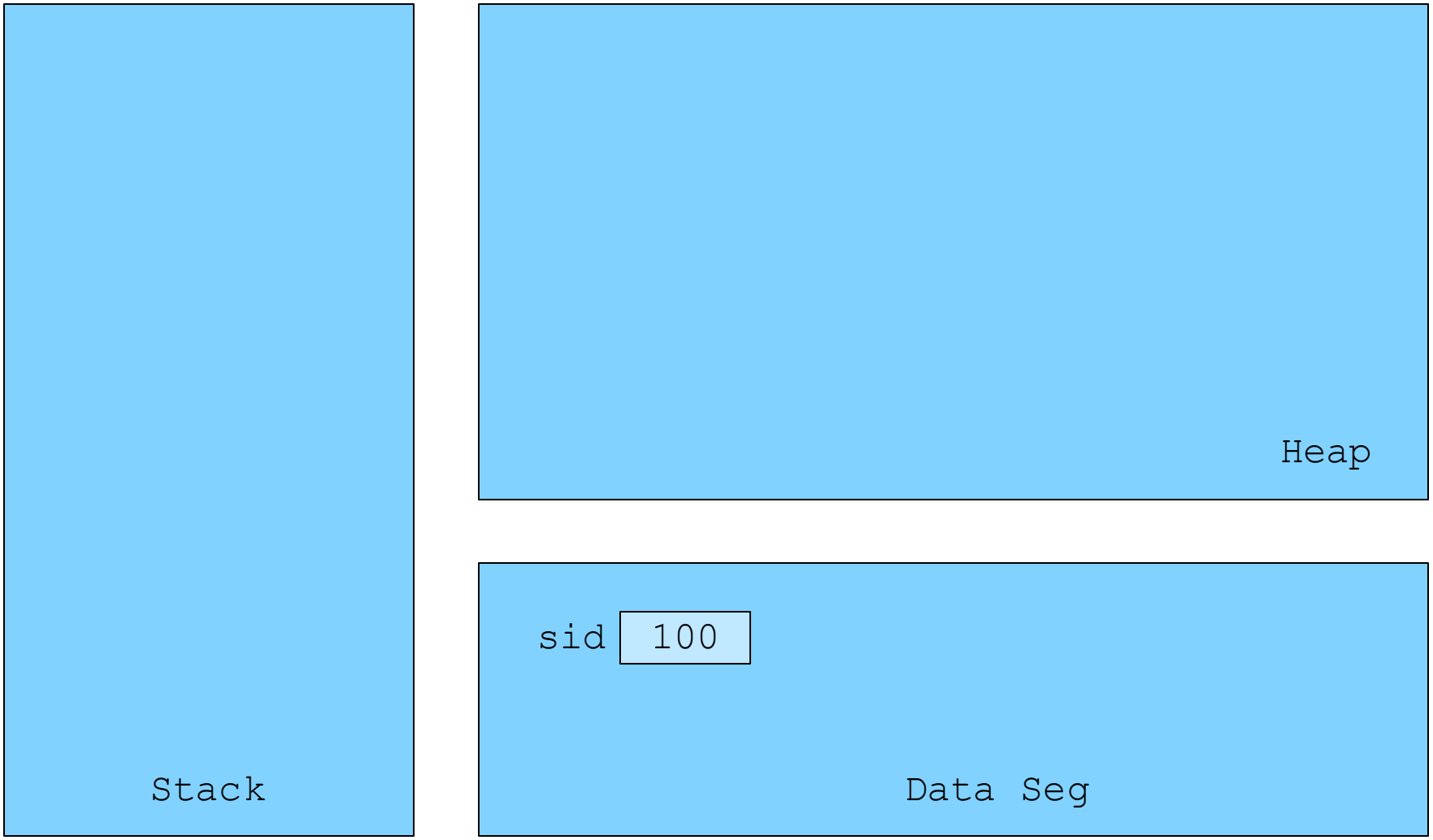
TestThis.java

The Keyword `static`

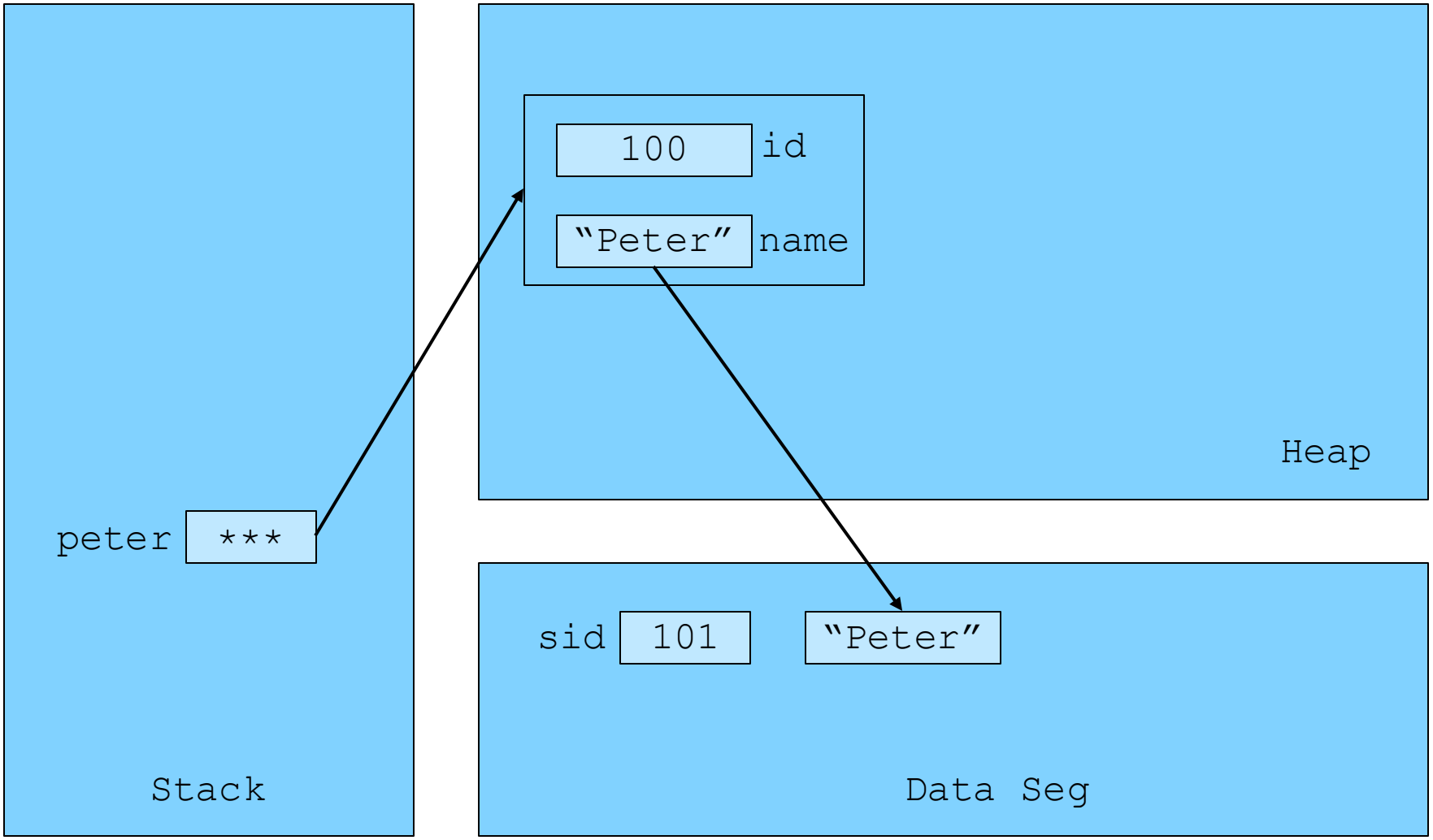
- When a variable or method of a class is declared as `static`, it is associated with the class as a whole, rather than with each individual instance of that class
- The static can be:
  - Instance variable
  - Method
  - Block
  - Nested class

```
public class Student {
    private static int sid = 0;
    private String name;
    int id;
    Student(String name) {
        this.name = name;
        id = sid++;}

    public void info(){
        System.out.println("My name is "+name+" No."+id);}

    public static void main(String arg[]){
        Student.sid = 100;
        Student peter = new Student("Peter");
        Student emma = new Student("Emma");
        peter.info();
        emma.info();}
}
```

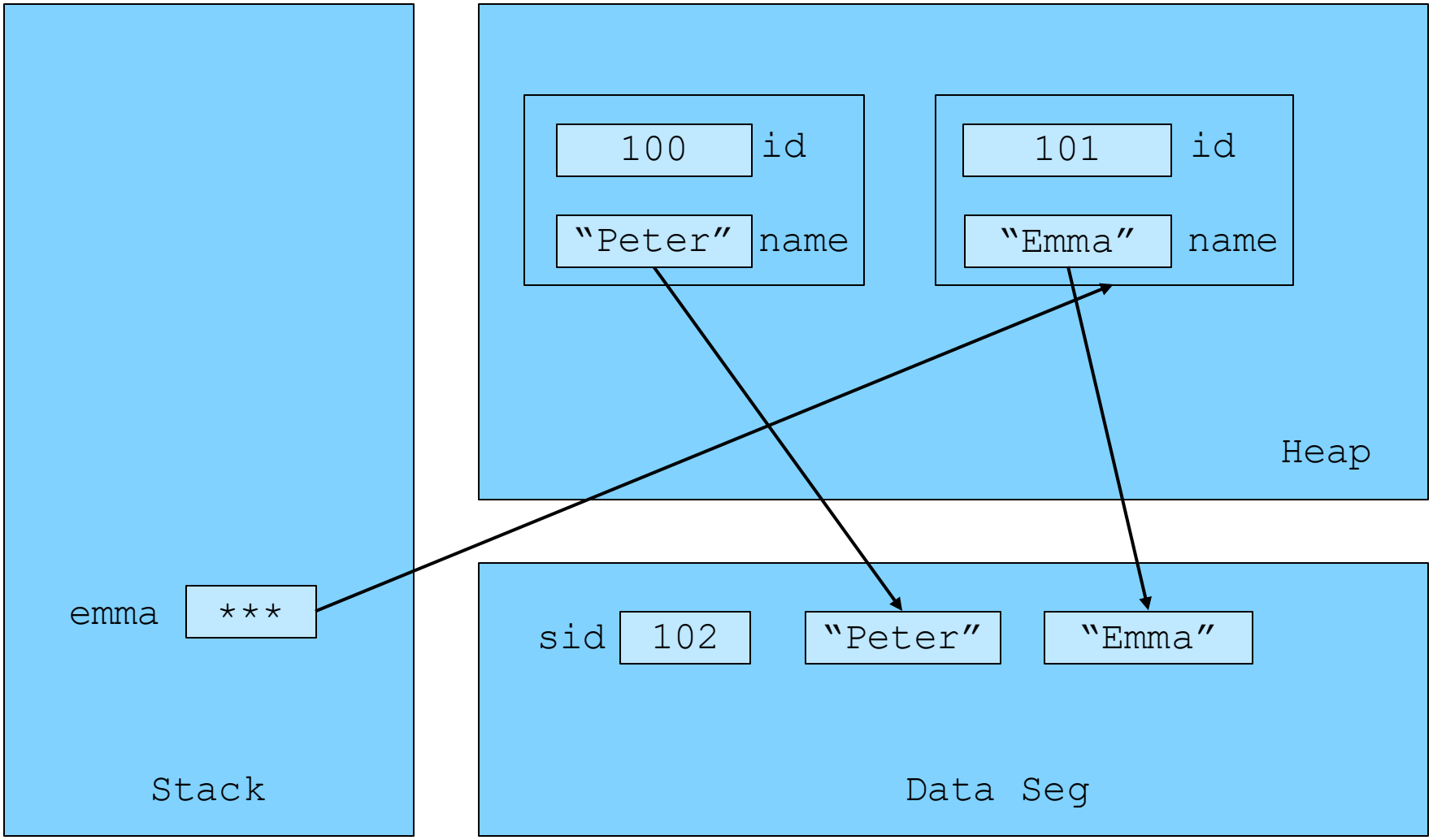
Student.java



Student.java



Student.java



Student.java

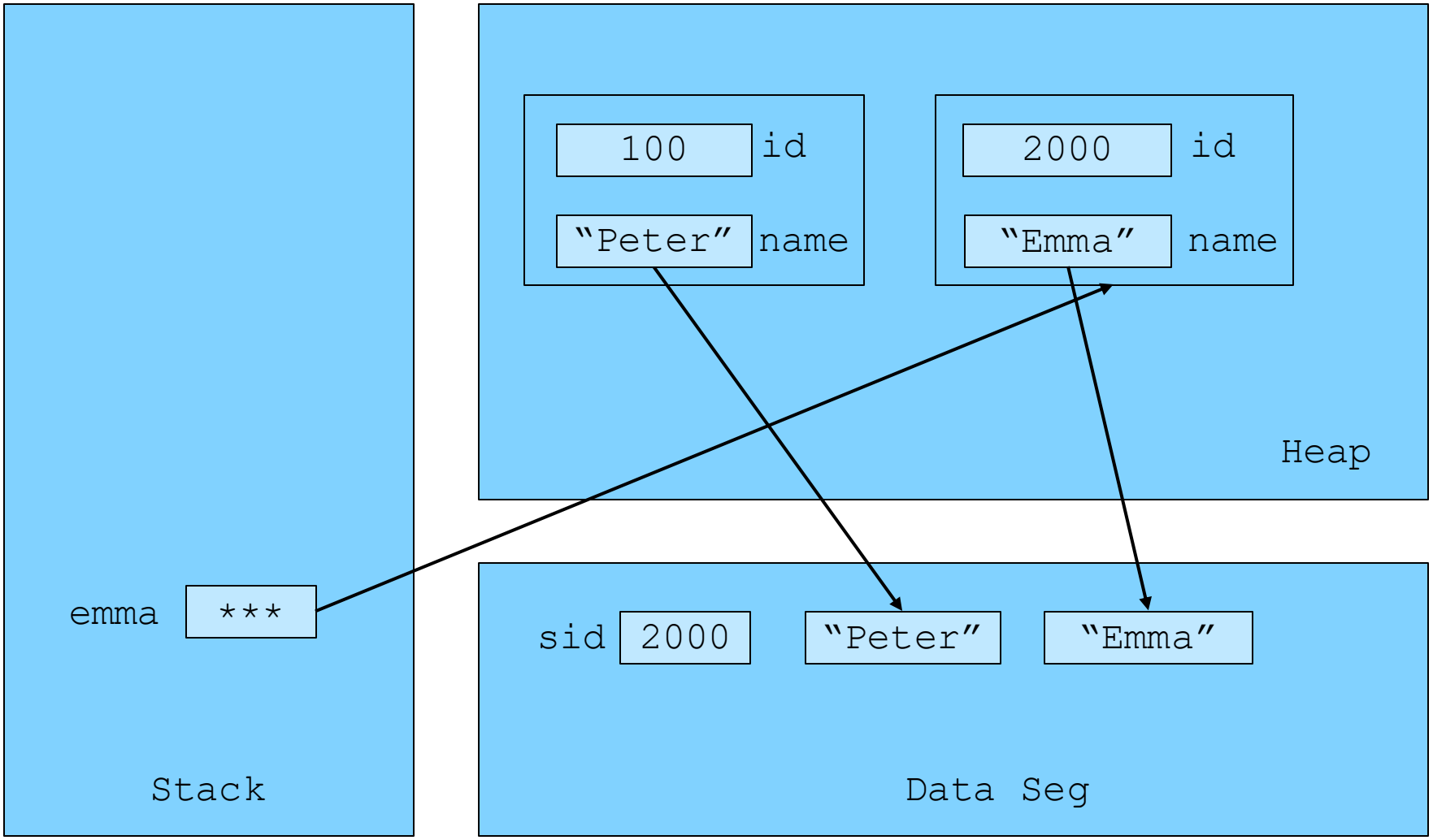
```
public class Student1 {
    private static int sid = 0;
    private String name;
    int id;
    Student1(String name) {
        this.name = name;
        id = sid++;
    }

    public void info(){
        System.out.println("My name is "+name+" No."+id);
    }

    public static void main(String arg[]){
        id = 100;
        sid = 2000;
        Student1 peter = new Student1("Peter");
        Student1 emma = new Student1("Emma");
        peter.info();
        emma.info();
    }
}
```

✘ ✓

Student1.java

```
public class Student2 {
    private static int sid = 0;
    private String name;
    int id;
    Student2(String name) {
        this.name = name;
        id = sid++;
    }

    public void info(){
        System.out.println("My name is "+name+" No."+id);
    }

    public static void main(String arg[]){
        Student2.sid = 100;
        Student2 peter = new Student2("Peter");
        peter.sid = 2000;
        Student2 emma = new Student2("Emma");
        peter.info();
        emma.info();
    }
}
```

Student2.java

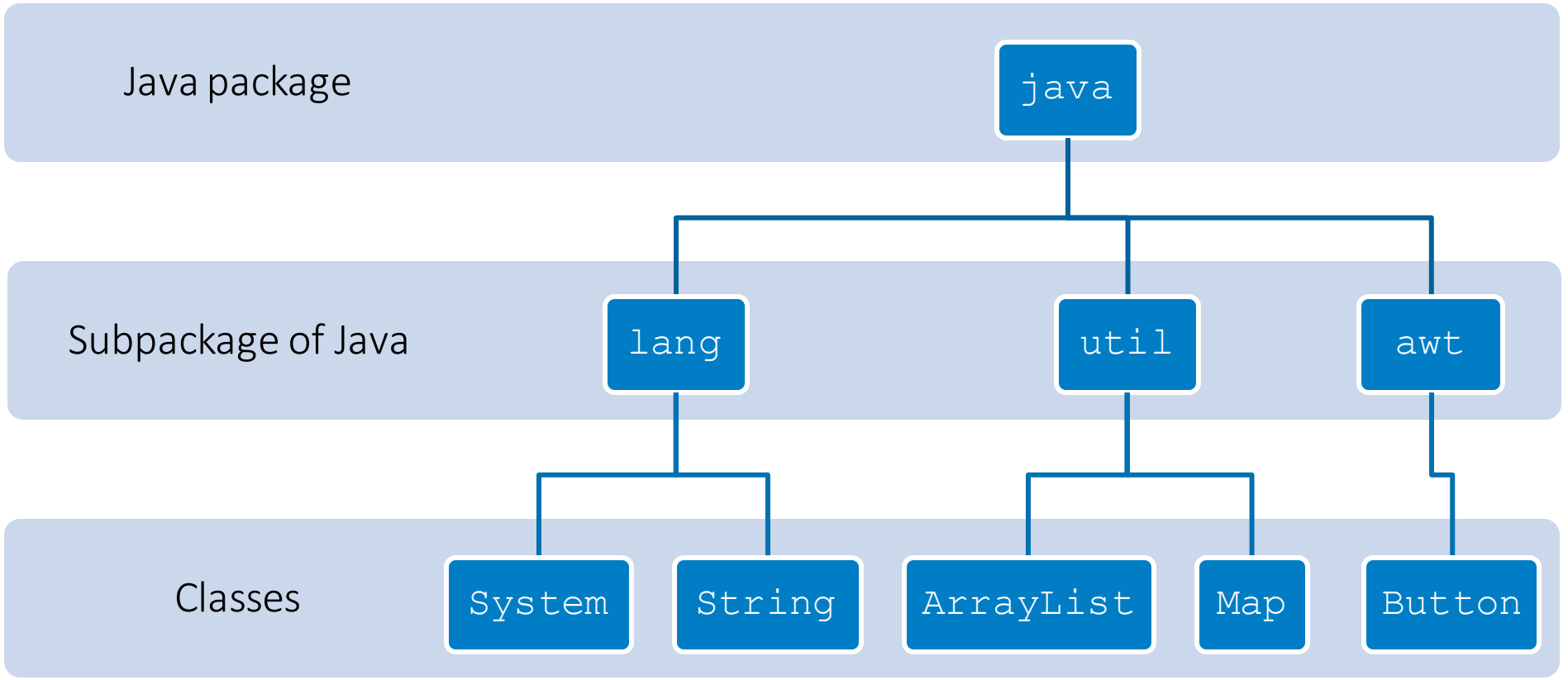


Student2.java

Package and Import

- A **package** is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of **packages** as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another
  - Java package is used to categorize the classes and interfaces so that they can be easily maintained
  - Java package provides access protection
  - Java package removes naming collision

Package and Import (cont'd.)



Package and Import (cont'd.)

- The **package keyword** is used to create a package in java
- If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages
- Format: `package pkg1.pkg2.pkg3…;`

Package and Import (cont'd.)

- Companies use their reversed Internet domain name to begin their package names
  - For example, ***com.example.mypackage*** for a package named ***mypackage*** created by a programmer at ***example.com***
- Include the region or the project name after the company name
  - For example, ***com.example.region.mypackage***

Package and Import (cont'd.)

- Each component of the package name corresponds to a subdirectory. So, if the Example company had a ***com.example.mypackage*** package that contained a Cat.java source file, it would be contained in a series of subdirectories like this:

  ....\com\example\mypackage\filename.java

- To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement
- Format: **import *pkg1.pkg2.pkg3….classname;***

  import *pkg1.pkg2.pkg3….*;*

```
package edu.mercy.chap2;

public class Cat{
   public void meow(){
      System.out.println("I can meow!");}
}
```

```
import edu.mercy.chap2.Cat;
```

`....\edu\mercy\chap2\Cat.class`

```
public class CreateCat{
   public static void main(String args[]){
      Cat c = new Cat();
      c.meow();
      //edu.mercy.chap2.Cat c = new edu.mercy.chap2.Cat();
   }
}
```

Cat.java

CreateCat.java

Access Control Modifiers

- The `public` class modifier designates that all classes may access the defined aspect
- The `protected` class modifier designates that access to the defined aspect is only granted to classes that are designated as subclasses of the given class through inheritance or in the same package
- The `private` class modifier designates that access to a defined member of a class be granted only to code within that class

Access Control Modifiers (cont'd.)

Modifier	Within class	Within package	Outside package by subclass only	Outside package
`private`	√			
`default`	√	√		
`protected`	√	√	√	
`public`	√	√	√	√

Inheritance

- A mechanism for a modular and hierarchical organization is **inheritance**
- This allows a new class to be defined based upon an existing class as the starting point
- The existing class is typically described as the **base class**, parent class, or superclass, while the newly defined class is known as the **subclass** or child class
- There are two ways in which a subclass can differentiate itself from its superclass:
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method
  - `@override` Indicates that a method declaration is intended to override a method declaration in a superclass
  - A subclass may also extend its superclass by providing brand new methods

Inheritance (cont'd.)

- Format:

```
Modifier class Name extends superclass {}
```

Inheritance and Constructors

- Constructors are never inherited in Java; hence, every class must define a constructor for itself
- All of its fields must be properly initialized, including any inherited fields
- The first operation within the body of a constructor must be to invoke a constructor of the superclass, which initializes the fields defined in the superclass
- A constructor of the superclass is invoked explicitly by using the keyword `super` with appropriate parameters
- If a constructor for a subclass does not make an explicit call to `super` or `this` as its first command, then an implicit call to `super()`, the zero-parameter version of the superclass constructor, will be made

```
public class SuperClass {
   private int n;
   SuperClass() {System.out.println("SuperClass()");}
   SuperClass(int n) {
      System.out.println("SuperClass's n: " + n);
      this.n = n;}
}

public class SubClass extends SuperClass{
   private int n;
   SubClass (int n) {
      System.out.println("SubClass's n: " + n);
      this.n = n;}
   SubClass () {
      super(300);
      System.out.println("SubClass()");}

   public static void main(String[] args) {
      SubClass sc1 = new SubClass();
      SubClass sc2 = new SubClass(400);}
}
```

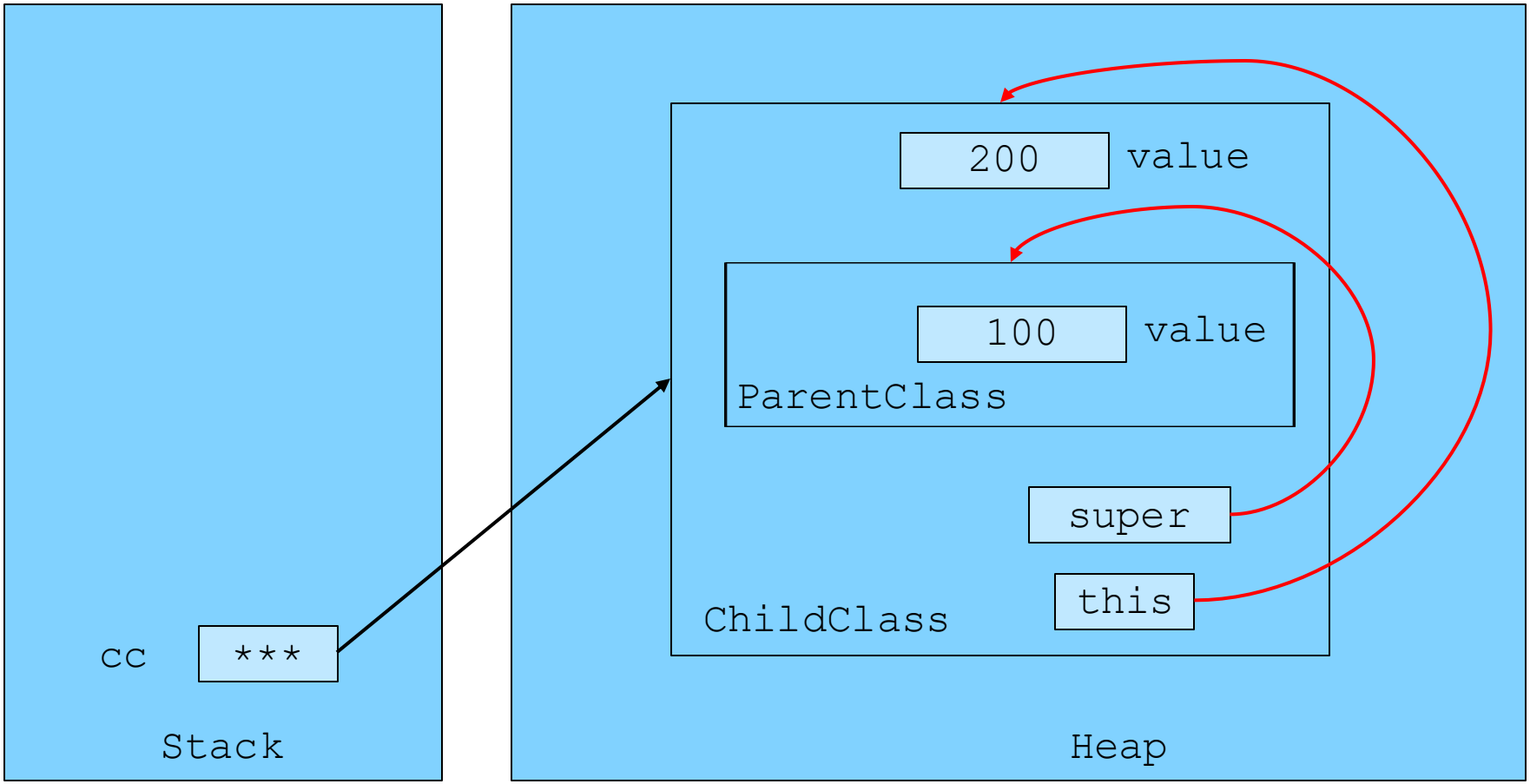
SuperClass.java

SubClass.java

```
public class ParentClass {
   public int value;
   public void f() {
      value = 100;
      System.out.println(“ParentCalss.value = " + value);}
}

public class ChildClass extends ParentClass {
   public int value;
   /** @override **/
   public void f() {
      super.f();
      value = 200;
      System.out.println("ChildClass.value = " + value);
      System.out.println(super.value);
   }

   public static void main(String args[]) {
      ChildClass cc = new ChildClass();
      cc.f();}
}
```

ParentClass.java

ChildClass.java



ParentClass.java

ChildClass.java

The Keyword `final`

- The `final` keyword can be used to modify variables, methods, and classes
  - Variables: can be initialized as part of that declaration, but can never again be assigned a new value
    - Base type: constant
    - Reference type: always refer to as the same object even if that object changes its internal state
    - Instance variable: often declared as static as well
  - Method: cannot be overridden by a subclass
  - Class: cannot be subclassed

Object Casting

- Casting with Objects allows for conversion between classes and subclasses
- A widening conversion occurs when a type *T* is converted into a "wider" type *U*:
  - *T* and *U* are class types and *U* is a superclass of *T*
  - *T* and *U* are interface types and *U* is a superinterface of *T*
  - *T* is a class that implements interface *U*
- Example: `U u = new T(...);`

Narrowing Conversions

- A narrowing conversion occurs when a type T is converted into a "narrower" type S
  - T and S are class types and S is a subclass of T
  - T and S are interface types and S is a subinterface of T
  - T is an interface implemented by class S
- In general, a narrowing conversion of reference types requires an explicit cast
- Example: `S s = (S) o;`
- We cast an object reference o of type T into type S, provided the object o is referring to is actually of type S → `instanceof` operator → throw `ClassCastException`

Generics

- Java includes support for writing generic classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts
- The generics framework allows us to define a class in terms of a set of formal type parameters, which can then be used as the declared type for variables, parameters, and return values within the class definition
- Those formal type parameters are later specified when using the generic class as a type elsewhere in a program

Syntax for Generics

- Types can be declared using generic names:

```
public class Pair<A,B>{
   A first;
   B second;
   public Pair(A a, B b){
      first = a;
      second = b;
   }
   public A getFirst() {return first;}
   public B getSecond() {return second;}
}
```

- They are then instantiated using actual types:

```
Pair<String,Double> bid = new Pair<>("Apple",23.8);

bid = new Pair<String,Double>("Apple",23.8);
```

Interfaces and Abstract Classes

- The main structural element in Java that enforces an application programming interface (API) is an **interface**
- An interface is a collection of method declarations with no data and no bodies
- Interfaces do not have constructors and they cannot be directly instantiated
- When a class **implements** an interface, it must implement all of the methods declared in the interface
- An abstract class is a class that is declared `abstract`—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed

Exceptions

- Exceptions are unexpected events that occur during the execution of a program
- An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer
- In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation
- An exception may also be **caught** by a surrounding block of code that "handles" the problem
- If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console

Catching Exceptions

- The general methodology for handling exceptions is a **try-catch** construct in which a guarded fragment of code that might throw an exception is executed
- If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch** block that contains the code to apply an appropriate resolution
- If no exception occurs in the guarded code, all **catch** blocks are ignored

```
try {
        guardedBody;
} catch (exceptionType1 variable1) {
        remedyBody1
} catch (exceptionType2 variable2) {
        remedyBody2
} …
```

Throwing Exceptions

- Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object
- This is done by using the **throw** keyword followed by an instance of the exception type to be thrown
- It is often convenient to instantiate an exception object at the time the exception has to be thrown. Thus, a throw statement is typically written as follows:

```
throw new exceptionType(parameters);
```

where *exceptionType* is the type of the exception and the parameters are sent to that type's constructor

The throws Clause

- When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method
- The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual **throw** statement)
- For example, the `parseInt` method of the Integer class has the following formal signature:

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

Nested Classes

- Java allows a class definition to be nested inside the definition of another class
- The main use for nesting classes is when defining a class that is strongly affiliated with another class
  - This can help increase encapsulation and reduce undesired name conflicts
- Nested classes are a valuable technique when implementing data structures, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure

Iterators

- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time

`hasNext()`	Returns `true` if there is at least one additional element in the sequence, and `false` otherwise
`next()`	Returns the next element in the sequence
`remove()`	Returns from the collection the element returned by the most recent call to `next()`. Throws an `IllegalStateException` if `next` has not yet been called, or if remove was already called since the most recent call to `next`

The `Iterable` Interface

- Java defines a parameterized interface, named `Iterable`, that includes the following single method:
  - `iterator()`: Returns an iterator of the elements in the collection
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the `iterator()` method
- Each call to `iterator()` returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection

For-Each Loops

- Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax:

```
for (elementType name : collection) {
        loopBody;
}
```

is equivalent to:

```
Iterator<elementType> iter = collection.iterator();
while (iter.hasNext()) {
        elementType variable = iter.next();
        loopBody;
}
```

Nested Classes

- Java allows a class definition to be nested inside the definition of another class
- The main use for nesting classes is when defining a class that is strongly affiliated with another class
  - This can help increase encapsulation and reduce undesired name conflicts
- Nested classes are a valuable technique when implementing data structures, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure

Summary

- Object-oriented programming
- Class, object, reference, constructor, and method overloading
- The `this` and `static` keywords
- Package and import
- Access control modifier
- Inheritance, method overriding, and the `final` and `super` keyword
- Polymorphism, object casting
- Abstract class, interface
- Exception
- Nested class
- Iterable, Iterator