Introducing Java

CSC207 Summer 2019

Agenda

- Basic Java Introduction (Learning the tool)
- Object Oriented Programming
- Announcements & Survey review

- Next Week: Git

Java Naming Conventions

- The Java Language Specification recommends these conventions
  - Generally: Use `camelCase` not `pothole_case`.
- **Class name**: A noun phrase starting with a capital.
- **Method() name**: A verb phrase starting with lower case.
- **Instance variable**: A noun phrase starting with lower case.
- **Local variable or parameter**: ditto, but acronyms and abbreviations are more okay.
- **Constant**: all uppercase, pothole_case.
  - E.g.: MAX_ENROLMENT

JavaDoc

- External Documentation (JavaDocs) vs. Internal Documentation (Comments)
- Like a Python docstring, but more structured, and placed above the method, classes, and variables.

```
/**
 * This method takes x and y, does something with it, and
 * returns the sum.
 *
 * @param x  The double to add
 * @param y  The integer to add
 *
 * @return The sum of x and y
 *
 * @throws PiException  If pi is not 22/7 today.
 * @see Integer
 */
public void sums_of_nums(double x, int y) { ... }
```

- starts with /**, not /*
- Classes: **@author**, **@version**
- Methods: **@param**, **@return**, **@returns**, **@see**

- This is where the Java API documentation comes from!
- In IntelliJ: Tools → Generate JavaDoc

Constructors

- A constructor has:
  - the same name as the class
  - no return type (not even void)
- A class can have multiple constructors, as long as their signatures are different.
- If you define no constructors, the compiler supplies one with no parameters and no body.
- If you define any constructor for a class, the compiler will no longer supply the default constructor.

`this`

- `this` is an instance variable that you get without declaring it.
- It’s like `self` in Python.
- Its value is the address of the object whose method has been called.

Instance Variables

```
public class Circle {
    private String radius;
    private static final double PI_NUMBER = 3.14;
    }
```

- radius is an instance variable. Each object/instance of the Circle class has its own radius variable.
- Using *static* variable will make them class-level variable (all instances share the same value)

Defining methods

- A method must have a return type declared. Use void if nothing is returned.

- The form of a return statement:

```
      return expression;
```

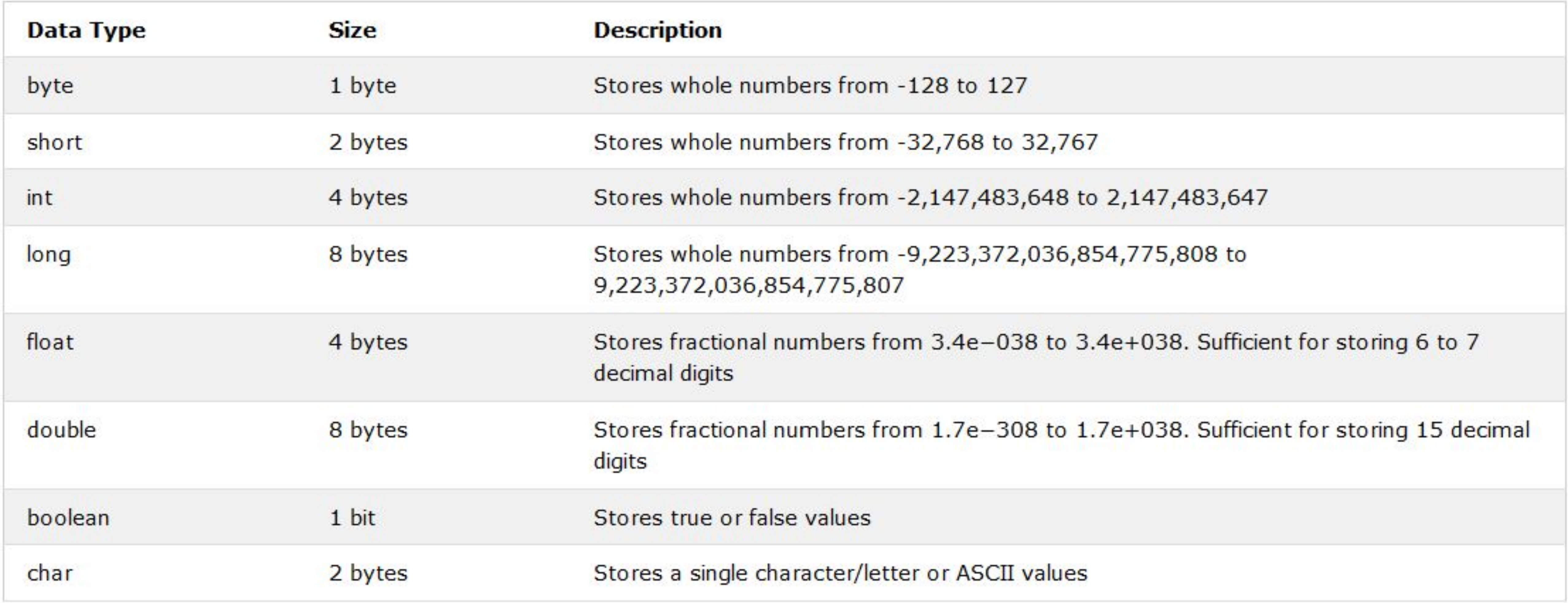
  If the expression is omitted or if the end of the method is reached without executing a return statement, nothing is returned.

- Must specify the accessibility. For now:
  `public` - callable from anywhere
  `private` - callable only from this class

- Variables declared in a method are local to that method.

Variable Types

Primitive types



Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers from 3.4e−038 to 3.4e+038. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers from 1.7e−308 to 1.7e+038. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Everything else is an object (referenced)

Autoboxing/Unboxing

- The automatic conversion Java makes between primitive types and their respective object wrappers
- It makes code clean!
- e.g. `Integer i = 5;`
- **Autoboxing**: primitive -> wrapper
- **Unboxing**: wrapper -> primitive
- Good for when we need object versions of things (e.g. in generics)

Primitive Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

Parameters

- When passing an argument to a method, you pass what’s in the variable’s box:
  - For class types, you are passing a reference. (Like in Python.)
  - For primitive types, you are passing a value. (Python can’t do anything like this.)
- This has important implications!
- You must be aware of whether you are passing a primitive or object.

Method Overloading

- Methods with the **same name** but **different parameters**
- e.g. four versions of Math.abs():
  - double abs(double d), float abs(float f), int abs(int i), long abs(long lng)
- Constructors are often overloaded

Casting for the compiler

- If we could run this code, Java would find the `charAt` method in `o`, since it refers to a `String` object:

```
   Object o = new String(“hello”);
char c = o.charAt(1);
```

- But the code won’t compile because the compiler cannot be sure it will find the `charAt` method in `o`.
    - Remember: the compiler doesn’t run the code. It can only look at the type of `o`.
- So we need to cast `o` as a `String`:
- char c = ((String) o).charAt(1);

Casting in Other Contexts

- Java automatically converts:
  - byte → short → int → long → float → double
  - char → int and above
  - boolean → no other types
- Moving in the other direction requires a **cast**, which is like a promise to the compiler that you know what you are doing.
- When you cast, information is sometimes lost.
- Example of casting (and info lost): THIS CAN CAUSE PROBLEMS!

```
double x = -57.99;

int i = (int) x; // i = -57
```

To the demo…

Method Overloading
Constructor Overloading

Access Modifiers

- Classes can be declared public or package-private.
- Members of classes can be declared public, protected, package-protected, or private.

Modifier	Class	Package	Subclass	World
`public`	Yes	Yes	Yes	Yes
`protected`	Yes	Yes	Yes	No
`default (package private)`	Yes	Yes	No	No
`private`	Yes	No	No	No

Conventions

- Make all non-final instance variables either:
    - *private*: accessible only within the class, or
    - *protected*: accessible only within the package.
- When desired, give outside access using “getter” and “setter” methods.

- [A final variable cannot change value; it is a constant.]

Object-Oriented Programming

- What you have seen so far…
  - Java
  - Classes
  - Static
  - Casting
  - Overloading
  - Autoboxing

- Now…
  - Encapsulation
  - Inheritance
    - Overriding
    - Shadowing
  - Polymorphism

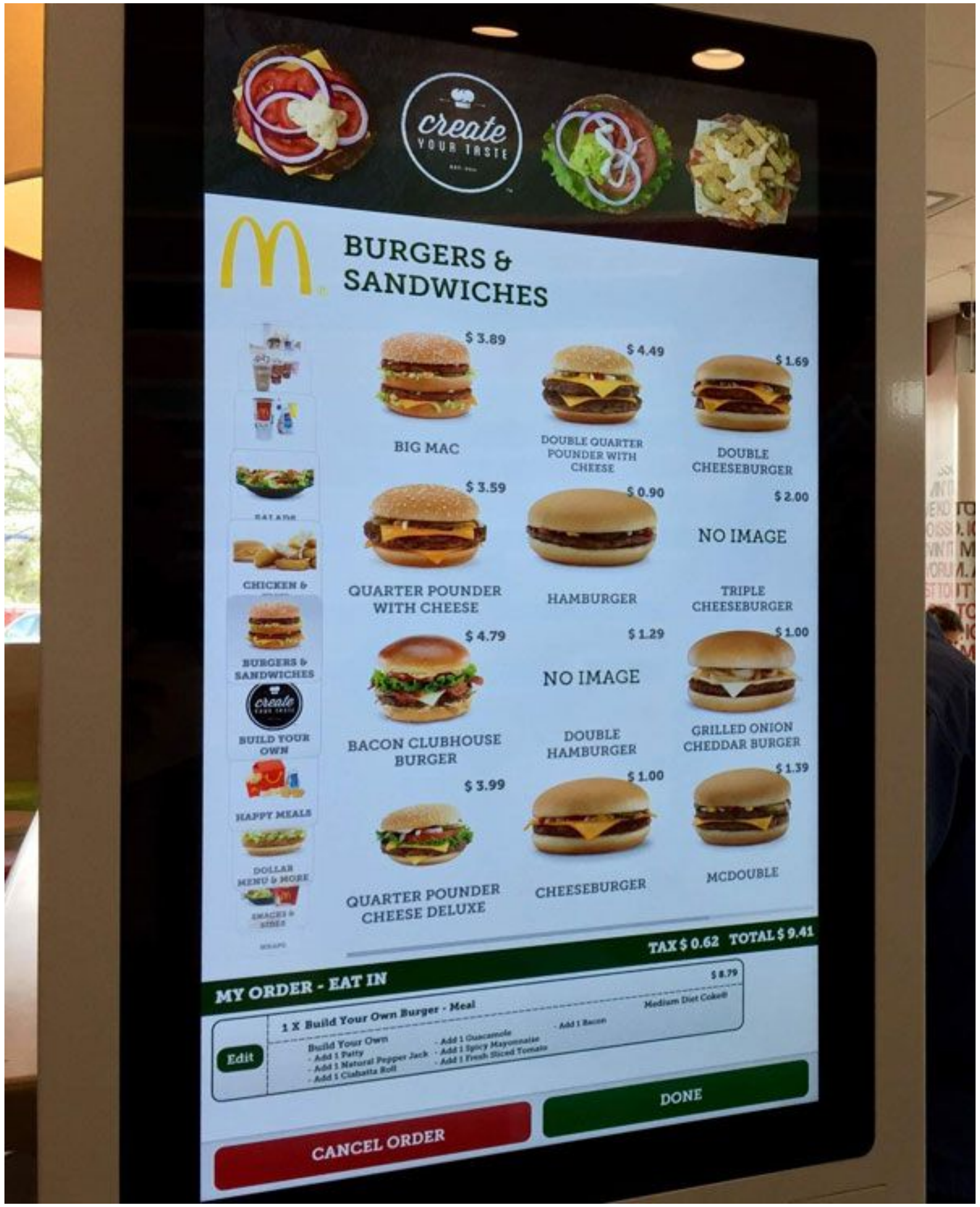
Why OOP?

- **Modularity**: code can be written and maintained separately, and easily passed around the system
- **Information-hiding**: internal representation hidden from the outside world
- **Code re-use**: others can implement/test/debug complex code, which you can then use in your own code
- **Pluggability and debugging ease**: “If a bolt breaks, you replace it, not the entire machine.”
- All together: SOLID design principles (more later...)

https://docs.oracle.com/javase/tutorial/java/concepts/object.html

Fundamental OOP concepts

- **Abstraction** — the process of distilling a concept to a set of essential characteristics.
- **Encapsulation** — the process of binding together data with methods that manipulate that data, and hiding the internal representation.
- The result of applying abstraction and encapsulation is (often) a class with instance variables and methods that together model a concept from the real world. (Further reading: what's the difference between Abstraction, Encapsulation, and Information hiding?)
- **Inheritance** — the concept that when a subclass is defined in terms another class, the features of that other class are inherited by the subclass.
- **Polymorphism** ("many forms") — the ability of an expression (such as a method call) to be applied to objects of different types.



TAX $ 0.62 TOTAL $ 9.41

MY ORDER - EAT IN

1 X Build Your Own Burger - Meal $ 8.79

Build Your Own
- Add 1 Patty
- Add 1 Natural Pepper Jack
- Add 1 Ciabatta Roll
- Add 1 Guacamole
- Add 1 Spicy Mayonnaise
- Add 1 Fresh Sliced Tomato
- Add 1 Bacon

Medium Diet Coke®

Encapsulation

- Think of your class as providing an abstraction, or a service.
  - We provide access to information through a well-defined interface: the public methods of the class.
  - We hide the implementation details.
- What is the advantage of this “encapsulation”?
  - We can change the implementation — to improve speed, reliability, or readability — and no other code has to change.

Inheritance Hierarchy

- All classes form a tree called the inheritance hierarchy, with `Object` at the root.
- Class Object does not have a parent. All other Java classes have one parent.
- If a class has no parent declared, it is a child of class `Object`.
- A parent class can have multiple child classes.
- Class `Object` guarantees that every class inherits methods `toString`, `equals`, and others.

Inheritance

- Inheritance allows one class to inherit the data and methods of another class.
- In a subclass, `super` refers to the part of the object defined by the parent class.
- Use `super.`*«attribute»* to refer to an attribute (data member or method) in the parent class.
- Use `super(`*«arguments»*`)` to call a constructor defined in the parent class.

Multi-part objects

- Suppose class `Child` extends class `Parent`.
- An instance of Child has:
  - a `Child` part, with all the data members and methods of Child
  - a `Parent` part, with all the data members and methods of Parent
  - a `Grandparent` part, … etc., all the way up to `Object`.
- An instance of `Child` can be used anywhere that a `Parent` is legal.
- But not the other way around.

Name Lookup

- A subclass can reuse a name already used for an inherited data member or method.
- Example:
  - class `Person` could have a data member `motto` and so could class `Student`. Or they could both have a method with the signature `sing()`.
  - When we construct

    ```
        x = new Student();
    ```

    the object has a `Student` part and a `Person` part.
  - If we say `x.motto` or `x.sing()`, we need to know which one we'll get!
- In other words, we need to know how Java will look up the name `motto` or `sing` inside a `Student` object.

Name Lookup Rules

- Calling a method: `expression.method(arguments)`
  - Java looks for method in the most specific, or bottom-most part of the object referred to by expression.
  - If it's not defined there, Java looks "upward" until it's found (else it's an error).
- Referencing an instance variable: `expression.variable`
  - Java determines the type of expression, and looks in that box.
  - If it's not defined there, Java looks "upward" until it's found (else it's an error).

Shadowing and Overriding

- Suppose class `A` and its subclass `AChild` each have an instance variable `x` and an instance method `m`.
- A’s m is **overridden** by `Achild’s m`.
  - This is often a good idea. We often want to specialize behaviour in a subclass.
- A’s x is **shadowed** by `Achild’s x`.
  - This is confusing and rarely a good idea.
- If a method must not be overridden in a descendant, declare it `final`.

Interlude on the Memory Model

Memory Model

- Stack: If no more memory → **java.lang.StackOverflowError**
- Heap: If no more memory → **java.lang.OutOfMemoryError**
    - Aside: JVM options:
        - -Xms: initial Java Heap size
        - -Xmx: maximum Java Heap size
        - -Xmn: the size of the heap
  - **Garbage collection** removes unreachable objects in the heap
- Let's see how they are used

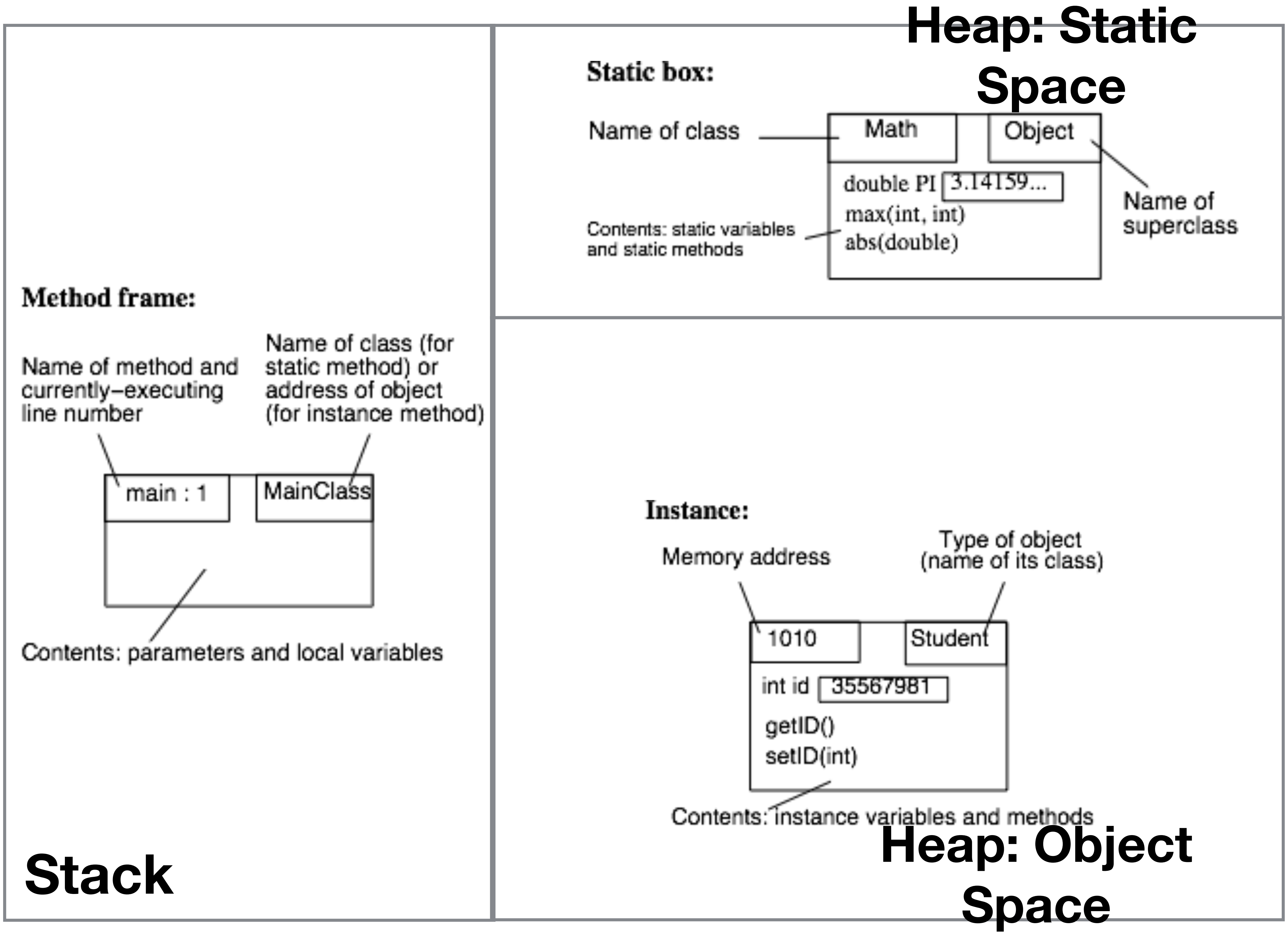
Sing()

- With the Java visualizer: https://goo.gl/xE3Ty6

```
public class UniversityDemo {
    public static void main(String[] args) {
        Person p1 = new Student();
        p1.sing();
    }
}

class Person {
    public void sing(){
        System.out.println("Caught in a bad romance!");
    }
}

class Student extends Person {
    public void sing() {
        System.out.println("No more paper, no more
books!");
    }
}
```



Method frame:

Name of method and currently–executing line number

Name of class (for static method) or address of object (for instance method)

main : 1

MainClass

Contents: parameters and local variables

Stack

Static box:

Name of class

Math

Object

double PI 3.14159...

max(int, int)

abs(double)

Contents: static variables and static methods

Name of superclass

Heap: Static Space

Instance:

Memory address

Type of object (name of its class)

1010

Student

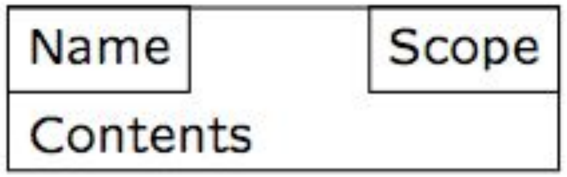
int id 35567981

getID()

setID(int)

Contents: instance variables and methods

Heap: Object Space

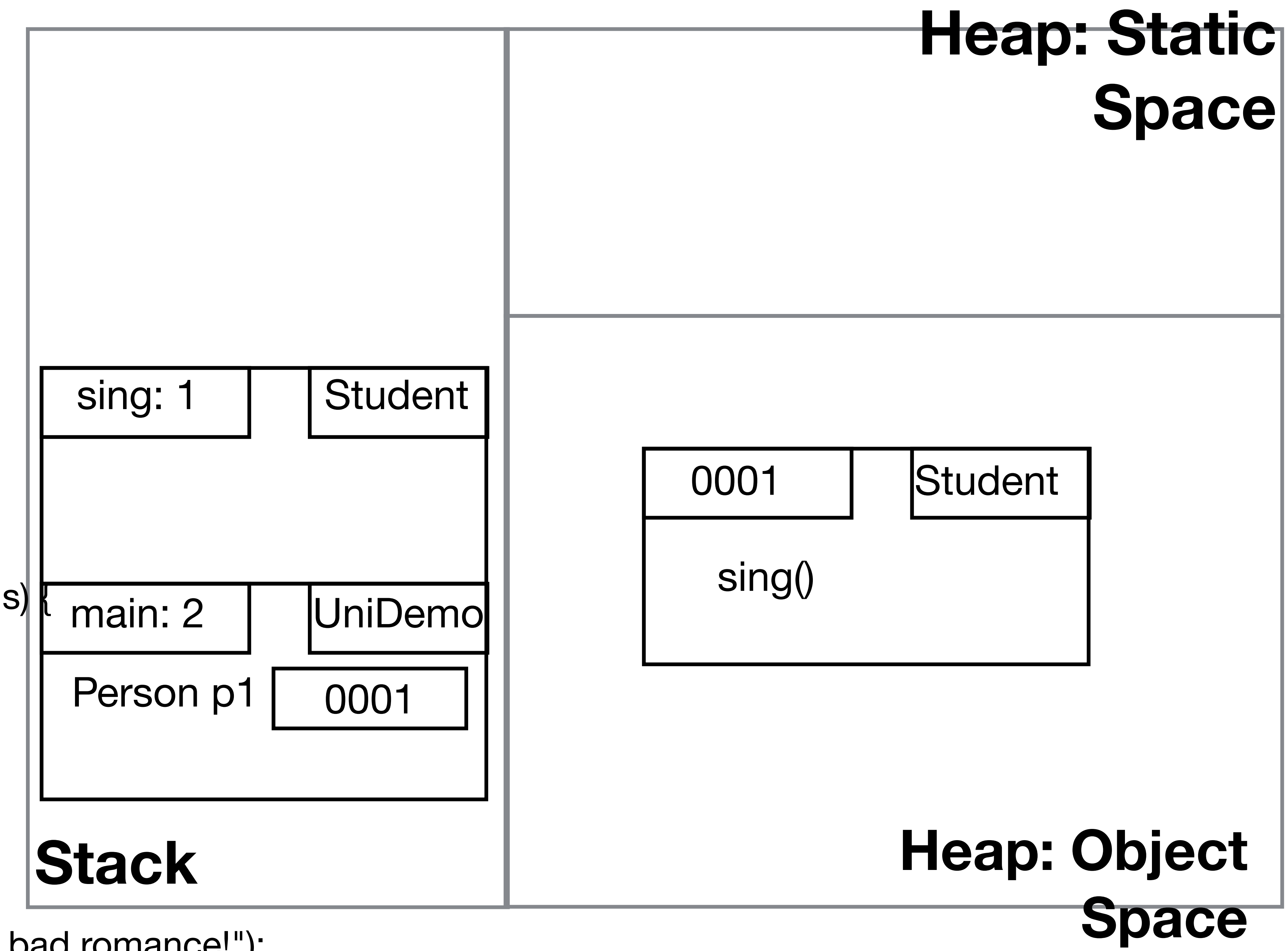


Name

Scope

Contents

= memory box



```
public class UniversityDemo {
    public static void main(String[] args) {
        Person p1 = new Student();
        p1.sing();
    }
}

class Person {
    public void sing(){
        System.out.println("Caught in a bad romance!");
    }
}

class Student extends Person {
    public void sing() {
        System.out.println("No more paper, no more books!");
    }
}
```

Polymorphism

- Definition: the ability of one thing to have multiple forms (i.e. inheritance, overloading, etc.), or one form to apply to several things (i.e. interfaces)
- Example: if `Student` and `Instructor` both extend `Person`

```
Person p;
p = new Person(“Lindsey”); // OK
p = new Student(“David”); // OK
p = new Instructor(“Paul”); // OK
```

https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html

Back to the demo…

Polymorphism