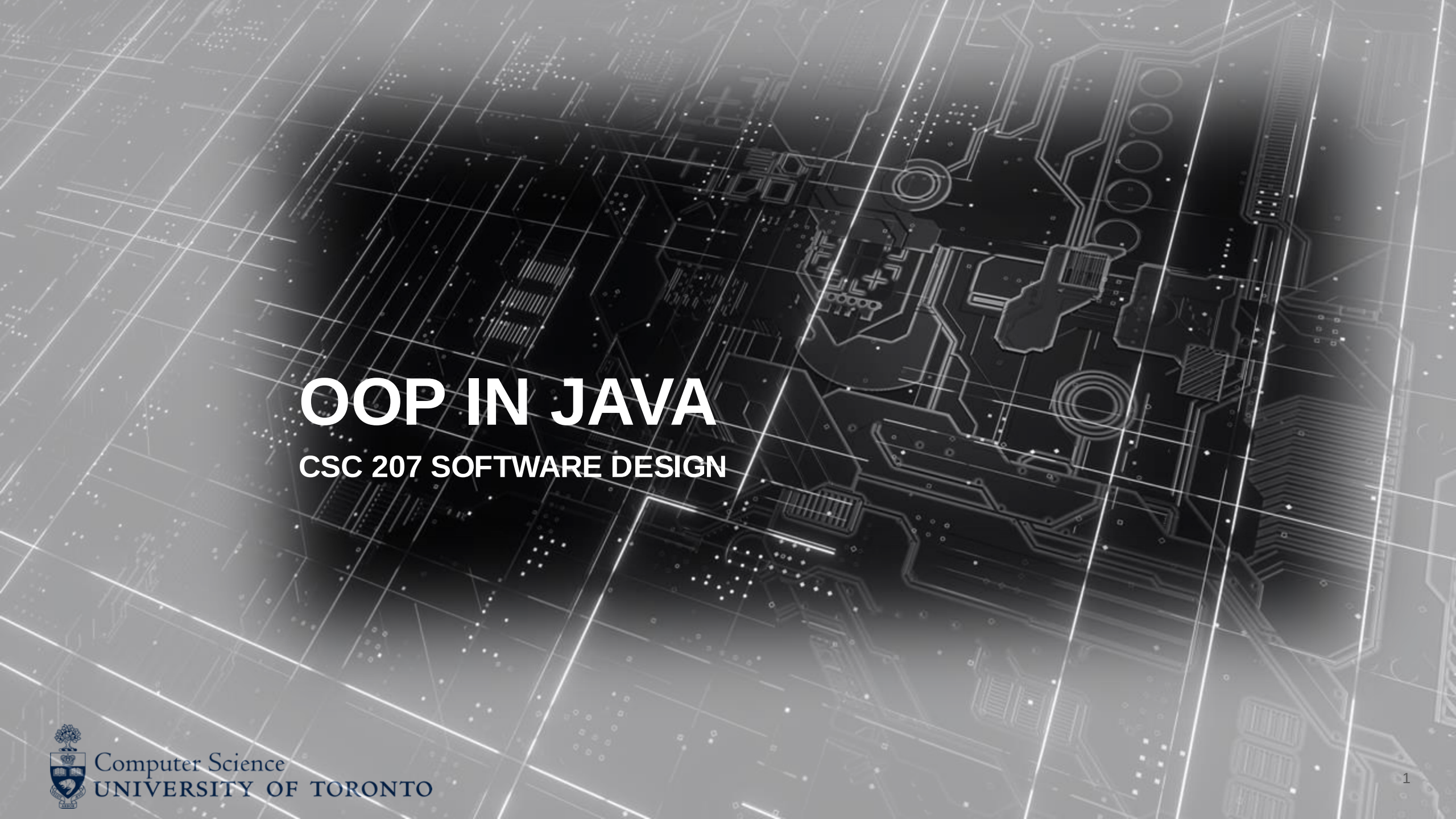


OOP IN JAVA

CSC 207 SOFTWARE DESIGN

Computer Science
UNIVERSITY OF TORONTO

LEARNING OUTCOMES

- Understand OOP in Java, including:
  - inheritance
  - encapsulation
  - a memory model for Java
  - method overriding
- Understand access modifiers in Java
- Identify concepts you want to review in the Java readings and practice in the self-assessment practice Java quizzes available on Quercus.

ENCAPSULATION

- Think of your class as providing 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 must change!*

INSTANCE VARIABLES AND ACCESSIBILITY

- If an instance variable is private, how can client code use it?
- Why not make everything public — so much easier!
- More about access modifiers at

  https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

CLASS DESIGN TIPS

- Everything should be as private as possible.
  - Always make instance variables private. Always!
- Create public methods only when necessary. These provide an API.
  - For each instance variable, decide whether to provide a public getter, a setter, or both. (You can ask IntelliJ to write these for you!)
- Look at how the class will be used. Are there additional methods that would make that easier?
- If you create any helper methods, make them private.

THE PROGRAMMING INTERFACE

- The "user" for almost all code is a programmer. That user wants to know:
  - ... what data your class manages
  - ... what actions it can take (methods)
  - ... what properties your object has (getter methods)
  - ... what guarantees your methods and objects require and offer
    - ... how they fail and react to failure
    - ... what is returned and what errors are raised

JAVADOC

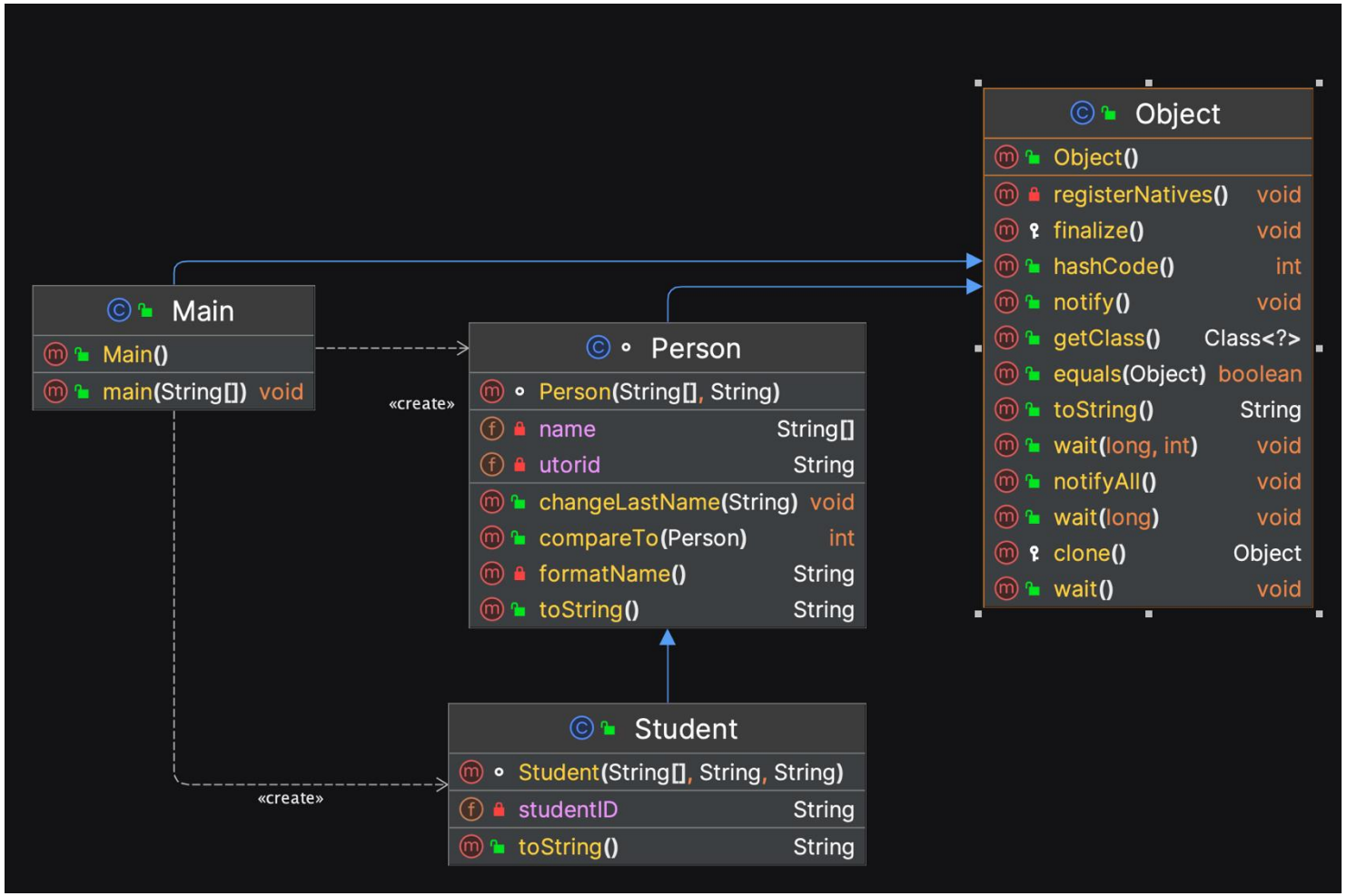
- Like a Python docstring, but more structured, and placed above the method.

```
/**
 * Replace a square wheel of diagonal diag with a round wheel of
 * diameter diam. If either dimension is negative, use a wooden tire.
 * @param diag  Size of the square wheel.
 * @param diam  Size of the round wheel.
 * @throws PiException  If pi is not 22/7 today.
 */
public void squareToRound(double diag, double diam) { ... }
```

- Javadoc is written for classes, member variables, and member methods.

- This is where the Java API documentation comes from!

INHERITANCE IN JAVA



This is a UML class diagram generated by IntelliJ Ultimate

Shows inheritance hierarchy

Shows class relationships

NOT a memory model!

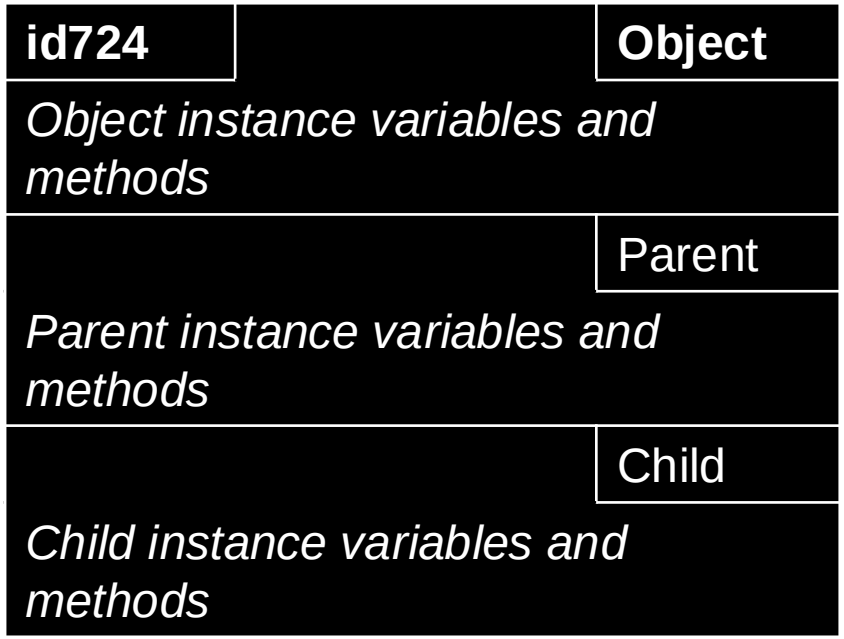
INHERITANCE HIERARCHY

- 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 some others.

JAVA MEMORY MODEL

MULTI-PART OBJECTS IN THE JAVA MEMORY MODEL

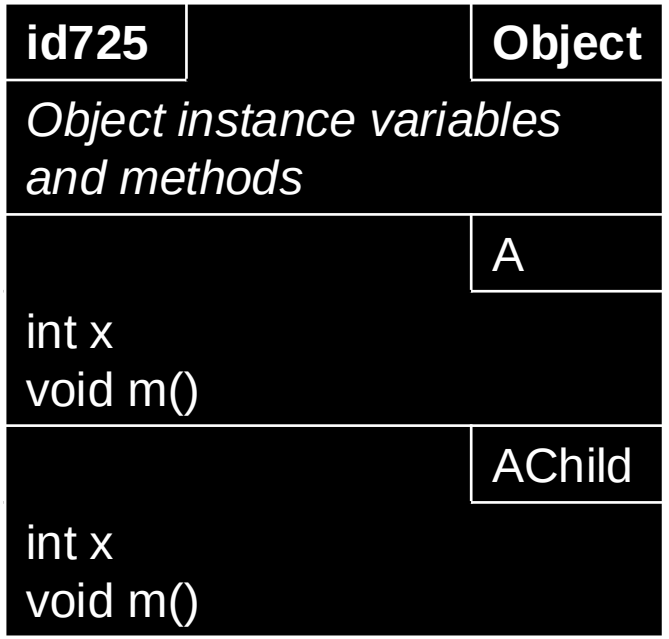
- Suppose class `Child` extends class `Parent`.
- Instance of class `Child`:
  - a `Child` part
  - a `Parent` part
  - a `Grandparent` part, … etc., all the way up to `Object`.



- An instance of `Child` can be used anywhere that a `Parent` is legal.
  - But not the other way around. (`Child` may have additional methods not in the `Parent` class.)

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`.
  - We often want to specialize behaviour in a subclass.
  - Java calls the **lowest** method in the object regardless of the reference type.
- `A`'s `x` is **shadowed** by `Achild`'s `x`.
  - Java uses the type of the reference to choose which to use.
  - Avoid public instance variables with the same name in a parent and child class.



If a method must not be overridden in a descendant, declare it `final`.

CASTING FOR THE COMPILER

- The Java compiler uses the type of the reference to determine whether a statement is valid.
- This code doesn't compile:

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

- The compiler does not keep track of object types, only variable types.

  Remember: the compiler doesn't run the code – it can only look at the type of `o` to determine whether it is legal.

- To do this, we need to cast `o` as a `String`:

```
char c = ((String) o).charAt(1);
```

- This is dangerous. Why?

CHECKING THE OBJECT TYPE

- At runtime, we can use operator `instanceof` to determine whether an object really is an instance of the specified type.

```
Object o = "Yo";
if (o instanceof String) {
    char c = ((String) o).charAt(1);
}
```

- What should the code do if `o` doesn't refer to a `String`?

WHAT HAPPENS WHEN WE CREATE AN OBJECT?

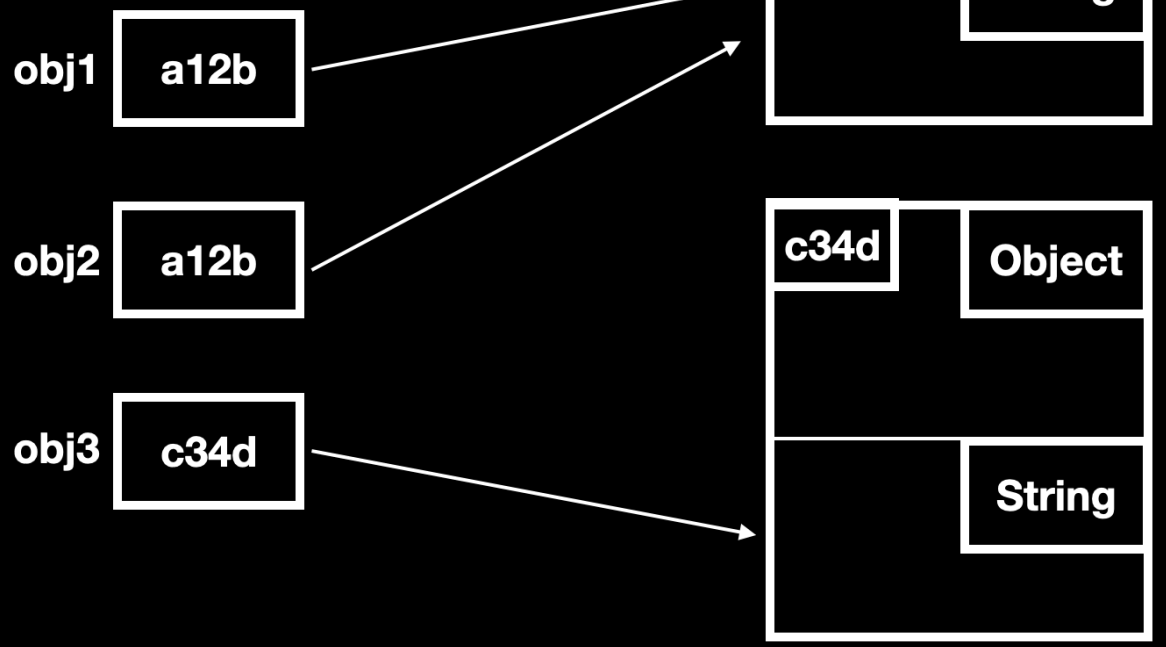
1. Keyword `new`
    1. instantiates an object
    2. Initializes all instance variables to their default values
        - 0 for `int`s, `false` for Booleans, etc., and `null` for class types.
        - Executes any direct initializations in the order in which they occur.
    3. Calls the appropriate constructor
        - The first line should be `super(`*arguments*`)`, or
        - If you don't call `super`, Java does it for you automatically

WHY CALL THE SUPER CONSTRUCTOR?

- Each object inherits necessary methods from the `Object` class, so that the Java language works the way we want.
- Example:
  ```
  Person p1 = new Person("abc", "123");
  System.out.println(p1);
  ```
  This should print something to the screen. But it will only do that if `Person` implements or inherits a `toString` method that can be called automatically by the `println` method.
- `Object` is therefore the top of every inheritance hierarchy in Java.
  All classes directly or indirectly extend `Object`.

Reference Types

```
String obj1 = "hello";
String obj2 = obj1;
String obj3 = "hi";
```



CASTING

- Consider this code:

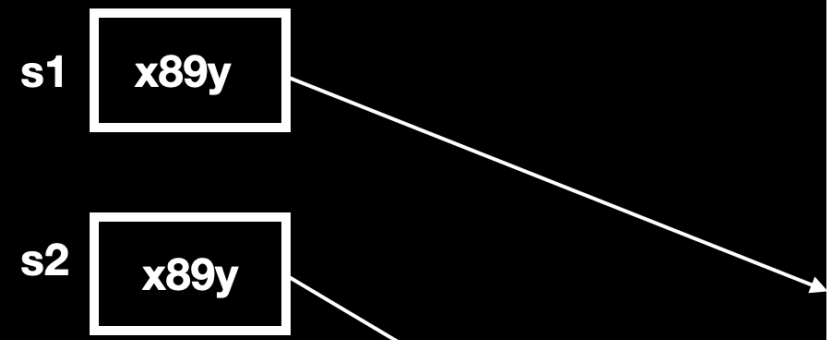
```
Person x = new Student(...);
Student y = (Student) x;
```

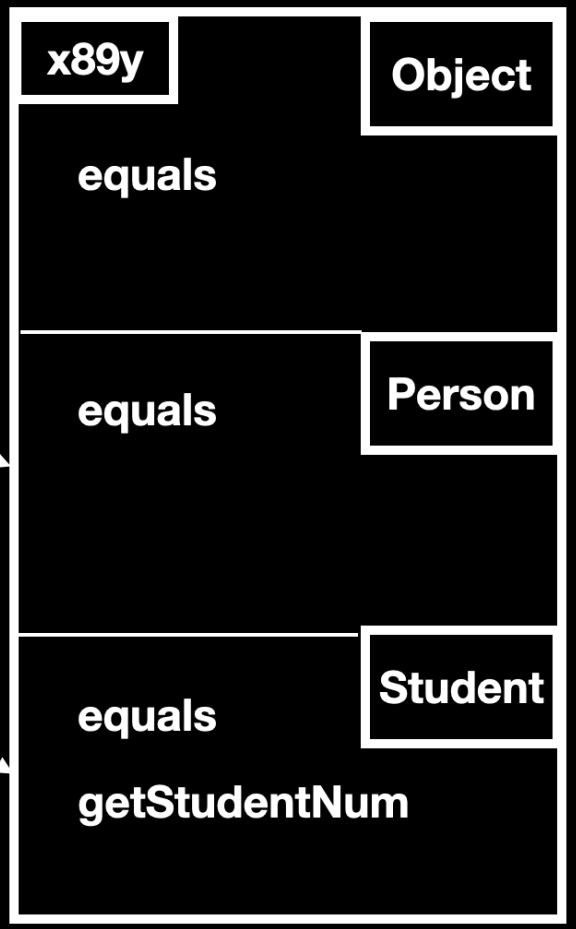
- Why do we need the word "`(Student)`"?
  - Without it, the compiler considers `x` to have type `Person`, so it cannot be assigned to variable `y` since the compiler doesn't know that `x` really does refer to a `Student` object.
  - With it, we cast `x` to have type `Student` for only that one line of code. Then it returns to having type `Person`.
- We call it casting when we change the type of an expression.
- We can only do that when the variable inherits or directly has the methods and variables of the type to which we are casting.
  - For example, `x` refers to an object that was created by calling the `Student` constructor. Since it has the variables and methods of a `Student`, we can cast it to `Student`. We cannot cast it to `Integer`, or `PartTimeStudent`, for example.

Casting

```
Person s1 = new Student("A B");

Student s2 = (Student) s1;
```



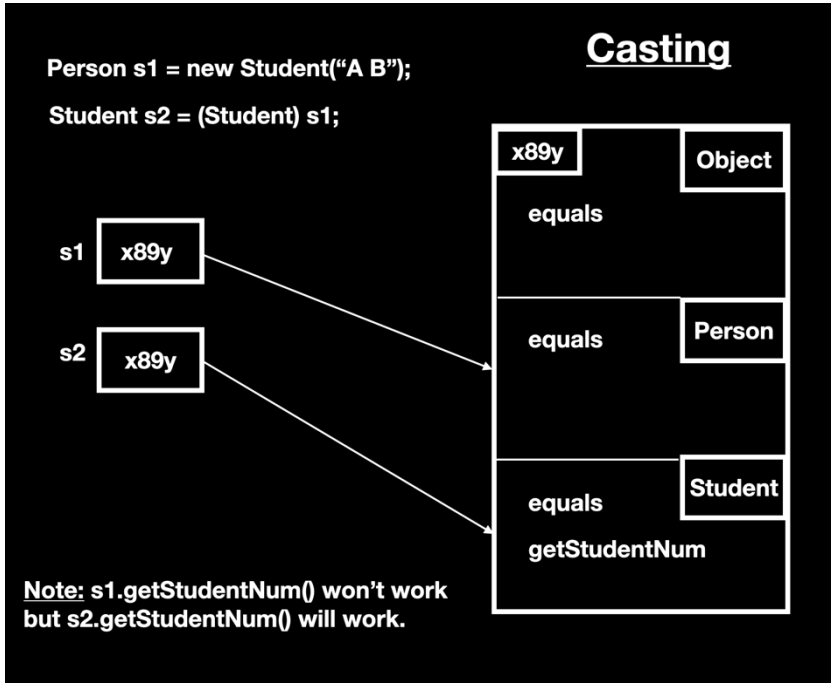


Note: s1.getStudentNum() won't work but s2.getStudentNum() will work.

JAVA LOOKUP RULES

- Consider:

```
Person p1 = new Student(…);
p1.moogah();
```



- If `moogah` is a method in the Person class, it will be executed.

- If `moogah` is NOT a method in the `Person` class, but it exists only higher in the inheritance hierarchy (example: in `Object`), than the inherited method will be executed.

- If `moogah` is an instance method that does not exist in `Person` or higher on the inheritance hierarchy, but it does exist in `Student`, the compiler will not know what to do. In this case, we have to cast `p1` back to student like this: `((Student)p1).moogah();`

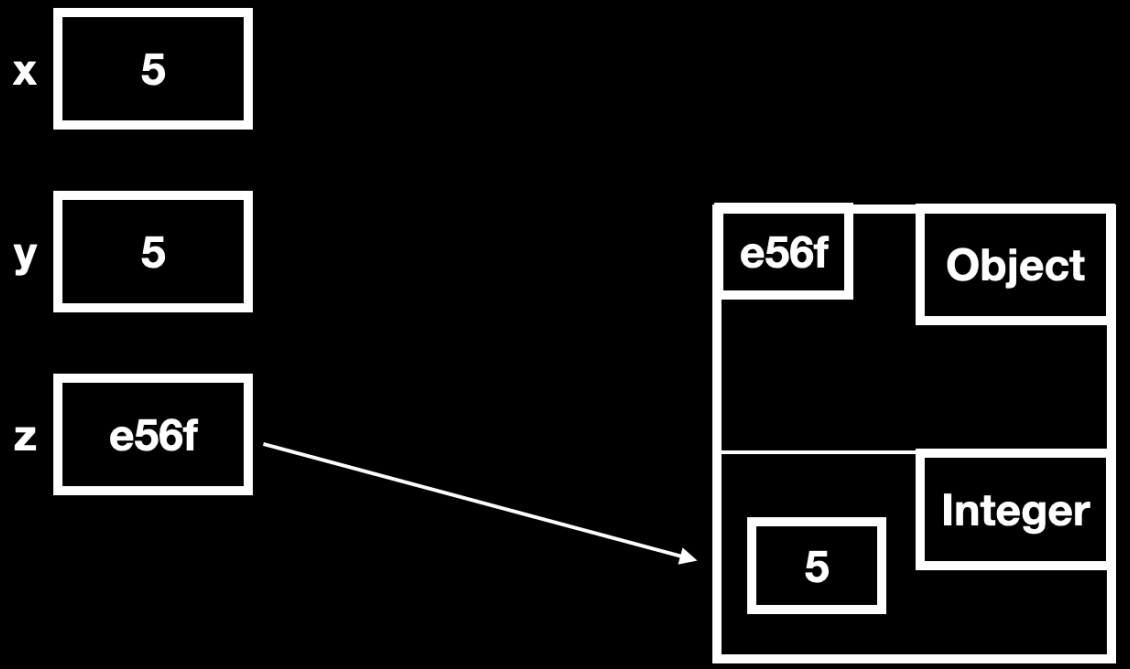
PRIMITIVE VS REFERENCE TYPES

- In the previous slide, `s1` stores the address where the instance of a class is actually stored.
- The fact that `s1` points at its "value" rather than directly storing the value means that `s1` has a reference type.

- On the next slide, we will see that `x` and `y` store their actual values, not an address that points to its value. This makes `int` a primitive type.
- You can read more about primitive types in Java here: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Primitive Types

```
int x = 5;
int y = x;
Integer z = new Integer(5);
```



WRAPPER CLASSES

- Each primitive type has a wrapper class.
  - For example, integers can be stored as a primitive value (`int a = 2;`) or as an Integer object (`Integer b = new Integer(2);`)
- Why do we need both?
  - Primitive values use less memory, but do not have constructors or methods
  - If you want to store a primitive value in a *generic* collection such as an `ArrayList`, we need a reference type to wrap the object in.
- Can we go between the two?
  - Yes! We can do this intentionally. Java does it automatically. For example:

```
int x = 2;
Integer y = new Integer(x);
```

AUTOBOXING AND UNBOXING

Sometimes Java can guess what you intended to do and will autobox (automatically make an object to store the primitive) or unbox (treat the value inside an object like its corresponding primitive)

```
int x = 2;
int y = 2;
Integer z = new Integer(2);
Integer w = z;
```

`x == y` will evaluate to true because `==` compares whatever is stored with the variable name. Since `x` and `y` are primitives, the value `2` is stored.

`z.equals(w)` will evaluate to `true` because the `Integer` equals method compares values.

`z == w` will evaluate to `true` because they both contain the same address of the `Integer` object that was created on the third line of code above.

`x == z` will evaluate to true because Java will automatically unbox `z` and compare its value instead of its address.

`x.equals(z)` will not run because Java will not know to autobox `x` into an `Integer` object. The parameter type is `Object`.