

**REVIEW**

# Review Topics

- Variables and data types
- Conditionals
- Loops
- Methods (including inputs and outputs)
- Pass by Value
- Writing Classes (including instance data variables, constructor, getters and setters, and toString)
- The equals method
- Inheritance and Overriding Methods
- Polymorphism
- null
- Coding Conventions

# **VARIABLES AND DATA TYPES**

# Variable Declaration

- A *variable* is a name for a location in memory
- Variables must be *declared*, meaning you state the:
  - variable type
  - variable name
- Examples:
  - `int total;`
  - `int count, sum;`
  - `String firstName;`

# Variable Initialization

- To use a variable, you must initialize it.
  - You can declare and initialize in separate statements or in the same statement.
- Examples:
  - `int sum = 0;`
  - `int n;`  
`n = 4;`
  - `int base = 32, mid, max = 149;`
  - `Student s = new Student(...);`

# Variable Type

- All Java variables are either a primitive type or an Object.
  - 8 primitive types
  - Object types defined by classes
- Java is a *strongly typed language*, meaning every variable has a declared type.
  - That declared type cannot change.
- Object variables also have an actual type.
  - This type could change (polymorphism!).

# Constants

- A *constant* is an identified that holds the same value throughout its existence.
  - The compiler will not let you change it.
- Use the `final` keyword to declare a constant.
  - By convention, constant names are in all caps and separated with an underscore.

```
final int MIN_HEIGHT = 60;  
final double AVG_TEMP = 98.6;
```

# Constants... Why?

- They give meaning to otherwise unclear literal values.
  - Example: `MAX_STUDENTS` has more meaning than 40.
- They facilitate program maintenance.
  - If a constant is used in multiple places, its value needs to only be updated in one place.
  - Example: if you can now hold 50 students instead of 40, you only have to edit the code that initializes `MAX_STUDENTS` instead of changing the number everywhere it appears.
- They formally establish that a value should not change, avoiding inadvertent errors by programmers.

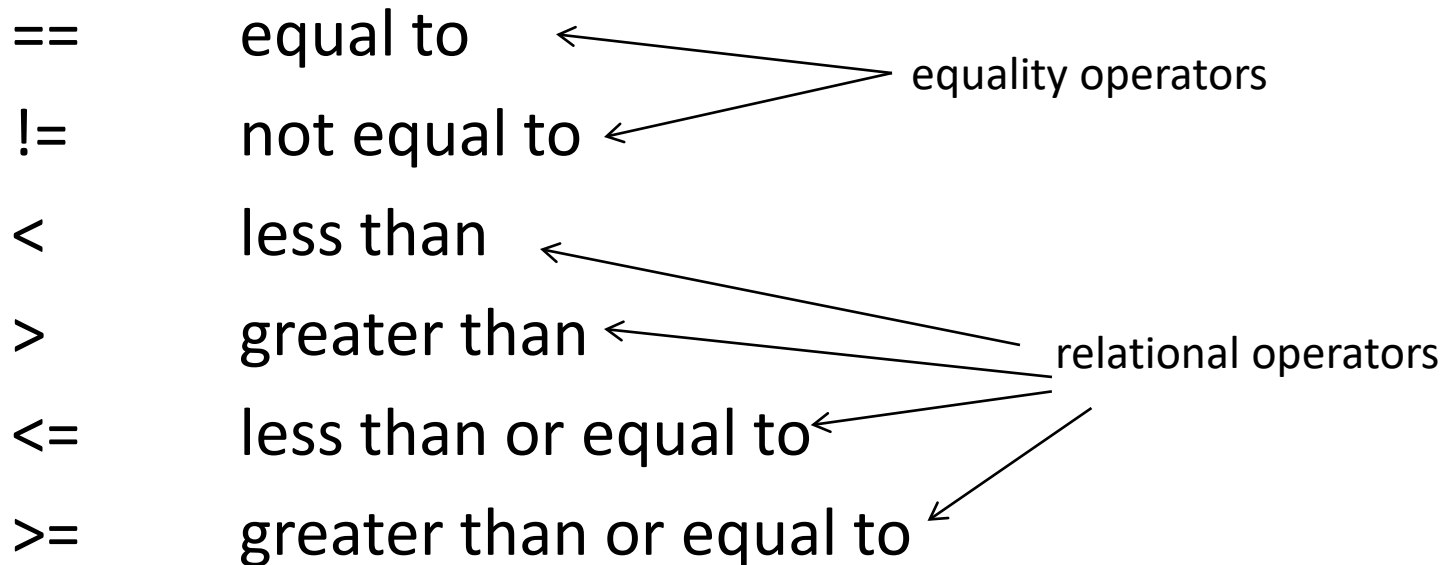


Before going to bed, a programmer put two glasses on his nightstand: one filled with water and one empty. Her husband asked her "why?" and she said, "The water is there in case I wake up at night and I feel thirsty." "What about the other one?" "Oh! This is in case I wake up at night and I am not thirsty!"

## **CONDITIONALS**

# Boolean Expressions

- A *boolean expression* evaluates to true or false.
- Boolean expressions often use an equality or relational operator to compare two values:



# Boolean Expressions (cont.)

- Boolean expressions can be combined using logical operators

! NOT

&& AND

|| OR

a	! a
true	false
false	true

a	b	a && b	a    b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

# Conditional Statements

- A *conditional statement* allows us to take different actions under different conditions.
- The simplest form is an if-statement.
  - If *boolean* is true, the code will *do something*.
  - If *boolean* is not true, nothing happens.

```
if (boolean) {  
    // do something  
}
```

# Indentation and Brackets

- Without brackets, one single statement after an if-statement will execute.
- With brackets, all statements inside will execute.
  - This is called a *block statement*.
- With or without brackets, best practice is to indent code within an if-statement.
  - Java doesn't care about this, but human readers do!

# The if-else Statement

- An else clause can be added to an if.
  - If *boolean* is true, *statement1* is executed.
  - If *boolean* is false, *statement2* is executed.
- One or the other statements is executed, but not both.

```
if (boolean) {  
    // statement1  
} else {  
    // statement2  
}
```

# The if-else Statement (cont.)

- Just like the if-statement, without brackets, one single line of code is execute.
- With brackets, all code inside the brackets will execute.

# Block Statements

- Variables declared inside of a block statement are local to that statement only.
  - They cannot be seen outside the brackets.

```
if (total > MAX)
{
    boolean error = true;
}
if(error)
    System.out.println("Error");
// COMPILER ERROR
```



# The if-else-if Statement

- You can add multiple conditions with else-if statements.
- Ending with a single else ensures that one of the statements is executed.

```
if (condition1) {  
    // do something1  
} else if (condition2) {  
    // do something2  
} else if (condition3) {  
    // do something3  
} else {  
    // do something4  
}
```

# Conditional Rules- What is Allowed

- a single if with
  - no else
- a single if with
  - a single else
- a single if with
  - any number of else-ifs
  - a single else
- a single if
  - any number of else-ifs
  - no else
- Conditionals are matched based on their *brackets*.
  - But indentation helps humans, too!

# The Conditional Operator

- A way to rewrite an if-else in a single statement

`condition ? expression1 : expression2 ;`

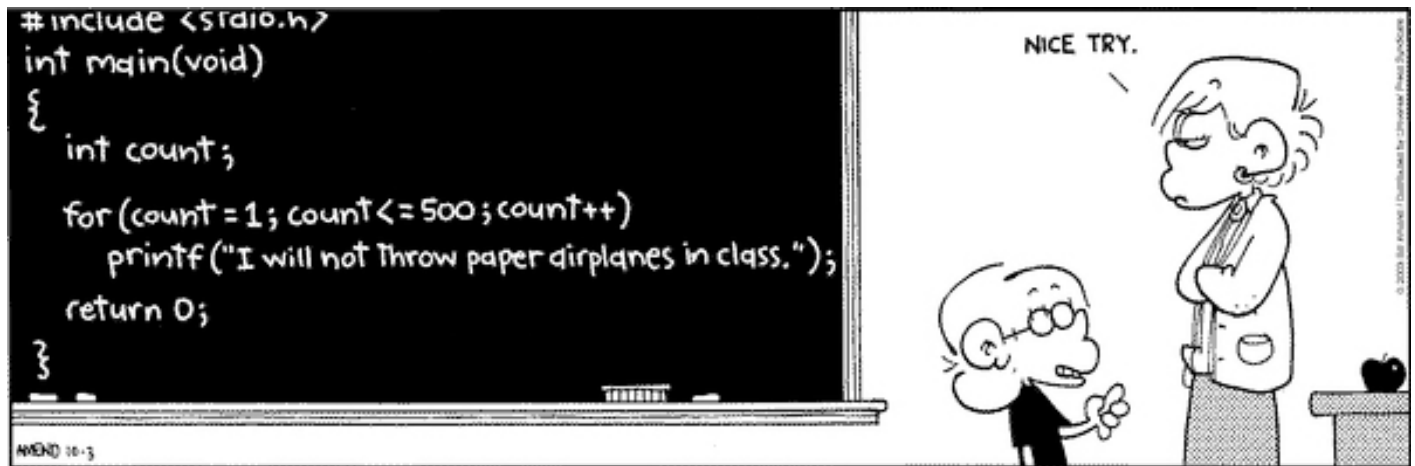
- If condition is true, expression1 is evaluated.
- If condition is false, expression2 is evaluated.

# Example

```
int larger =  
    num1 > num2 ? num1 : num2;
```

this is equivalent to:

```
int larger;  
if (num1 > num2)  
    larger = num1;  
else  
    larger = num2;
```



# LOOPS

# The while Loop

```
while (condition) {  
    statement;  
}
```

## 1. Check condition.

### a) If true:

- i. Execute Statement.
- ii. Return to Step 1.

### b) If false:

- i. Move on to the statement after the while loop.

# The while Loop (continued)

```
while (condition) {  
    statement;  
}
```

- As long as *condition* is true, keep executing the code inside (statement).
- Be sure that somewhere inside the loop the *condition* is set to false.
  - Otherwise the loop never ends!

# The while Loop (continued)

- The body of a while loop is executed zero or more times.
  - If the condition is false, the statement will never be executed.



# The while Loop

- The condition can be a complex boolean statement.

```
while (condition1 && condition2) {  
    statement;  
}
```

```
while (condition1 || condition2) {  
    statement;  
}
```

# Local Variables

- Variables declared inside the while loop are local only to that single iteration of the while loop.

```
while (condition) {  
    int n=0;  
    System.out.println(n) ;  
    ...  
}
```

# Uses of the while Loop

- You don't know how many times you want to repeat something ahead of time
- You want to repeat until some condition is reached
- You want to continue to obtain input until you get a *sentinel* value (to indicate input is done)
- You want to continue to obtain input until you get a valid value

# The do... while Loop

```
do {  
    statement;  
} while (condition);
```

1. Execute statement.
2. Check condition.
  - a) If true:
    - i. Return to Step 1.
  - b) If false:
    - i. Move on to the statement after the do-while loop.

# The do... while Loop (continued)

- The body of a do-while loop is executed **one** or more times.
- Which to use?
  - Do you want the condition evaluated *before* or *after* the code is executed?
  - Do you want the code executed always once or maybe executed never?

# The for Loop

```
for(initialization; condition; update) {  
    statement;  
}
```

1. Perform initialization.
2. Check condition.
  - a) If true:
    - i. Execute Statement.
    - ii. Perform update.
    - iii. Return to Step 2.
  - b) If false:
    - i. Move on to the statement after the for loop.

# The for Loop

- The initialization section can be used to declare a variable.
  - Note that this variable is then *local* to the loop- it cannot be seen outside of the loop.
- The update section can perform any calculation.
- Each section of the loop is optional.

# The for Loop (continued)

- A for loop is functionally equivalent to a while loop:

```
initialization;  
while (condition) {  
    statement;  
    update;  
}
```



# The for Loop (continued)

- The condition of a for loop is tested first.
- The body of a for loop is executed zero or more times.
  - If the condition is false, the statement will never be executed.

# Uses of the for Loop

- Executing statements a pre-defined number of times
- Iterating through collections (arrays, ArrayLists, etc.)

# Local Variables

- Variables declared inside the for loop are local to the for loop.

```
for(int i=0; i<n; i++) {  
    System.out.println(i);  
}
```

```
System.out.println(i); // ERROR!
```

# When to use which?

- In general, use a for-loop if you know in advance how many times you want to repeat the code.
- In general use a while-loop if you want to keep repeating code until some condition changes (something becomes true/false), but you don't know ahead of time when this will happen.

# Beware of Infinite Loops!

- The body of a while or do-while loop must eventually make the condition false.
- The update section (or body of) a for loop must eventually make the condition false.



# Nested Loops

- Loops can be nested such that the body of one loop contains another loop.
- For each iteration of the outer loop, the inner loop goes through a full execution.
  - Statements inside an inner loop will execute (outer \* inner) times.

# Example

- How many times is “Here” printed?

```
for(int i=0; i<20; i++) {  
    for(int j=0; j<40; j++) {  
        System.out.println ("Here");  
    }  
}
```

# The Enhanced `for` Loop (the “for each” Loop)

```
for (type varName : arrayName) {  
    statement;  
}
```

- Use to iterate through arrays
- Use to iterate through any collection *whose class implements Iterable*
  - This includes `ArrayList`, `LinkedList`, etc.



# The Enhanced `for` Loop (the “for each” Loop)

```
// scores is an int[]  
for(int score : scores) {  
    System.out.println("Score=" + score);  
}
```

– Read as: for each int in scores (called score), print the score

```
// catalog is a Course[]  
for(Course c : catalog) {  
    System.out.println(c.getTitle());  
}
```

– Read as: for each Course c in the catalog, print the title

# for-each Loops and for Loops

```
int[] nums = new int[10];  
for(int i=0; i<nums.length; i++)  
    nums[i] = i*5;
```

```
for(int i=0; i<nums.length; i++)  
    System.out.println(nums[i]);
```

```
for(int eachNum : nums)  
    System.out.println(eachNum);
```

**equivalent  
output**



# for-each Loops and for Loops

The for-each is the same as if you had a for-loop but declared a local variable inside the loop: `int eachNum = nums[i]`

```
for(int i=0; i<nums.length; i++) {  
    int eachNum = nums[i];  
    // do something with eachNum  
}
```

**equivalent!**

```
for(int eachNum : nums) {  
    // do something with eachNum  
}
```

# When to use which?


- Both are acceptable.
  - Although the for-each is gaining in popularity because many people find it easier to read and it eliminates the need for the (often meaningless) index variable.
  - If you don't care about the index value, use a for-each!
  - Oracle also now recommends its use:  
<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html>
- Anything you can do with a for-each can also be done with a regular for.
- But there are things you *can* do with a for-loop that you *cannot* do with a for-each.

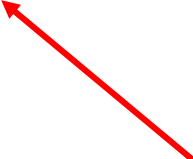
# What can't a for-each do?

- Access only *some* elements of the array
  - Example: `for(int i=0; i<array.length; i=i+2)`
    - Allows you to process every-other element in an array
    - Can't be done directly with a for-each
- Access the *index* of the element
  - Example: `for(int i=0; i<array.length; i++)`  
`System.out.println("The item at pos. " + (i+1) + " = " + array[i]);`
  - Allows you to fill an array based on the index
  - Since there is no index variable in a for-each, you can't do this directly
- *Update* the value stored in an array
  - Example: `for(int i=0; i<array.length; i++)`  
`array[i] = newValue;`
  - Allows you to update the contents of an array at a certain position
  - Since there is no index, there is no way to update at that position.

# Caution!

- You **cannot** update array contents with a for-each.

```
for (int eachNum : nums) {  
    eachNum = newValue;  mistake!!  
}
```

```
for (int i=0; i<nums.length; i++) {  
    int eachNum = nums[i];  
    eachNum = newValue;  that would be the  
equivalent of this-  
which does NOT  
change the array!  
}
```

# Bottom Line

- If you need to do one of the things on the previous slide, use a for loop.
- If you need to do anything else, it's your choice!
  - But consider the for-each!

# **METHODS**



# Method Header

- Method declaration begins with the method header:
  - visibility
  - return type
  - method name
  - formal parameters
    - type and name of each parameter
  - (static, when appropriate)
- Method body is surrounded by { }

# return Statement

- `void` methods do not return a value
- Methods not declared `void` must contain a `return` statement
  - Must return a value that matches the return type
- When you hit a return statement, control leaves the method and jumps back to the place where the method was invoked

# void Methods

- void methods do not return a value
- void methods can only be invoked as stand-alone statements (they execute and complete on their own)
- You cannot put a void method invocation inside another expression.
  - COMPILER ERROR:  
`System.out.println(myObject.printInfo());`
  - COMPILER ERROR: `int val = myObject.printInfo() + 5;`

# Valued Methods

- Valued methods return a value of a specific data type
- The return type can be primitive or Object
- You can use or ignore the returned value
- Examples:
  - `myObject.calculate();` // return value is ignored
  - `System.out.println(myObject. calculate());` // return value is printed
  - `int val = myObject. calculate();` // return value is stored in a local variable
  - `int val = myObject. calculate() * 2;` // return value is used in an expression and stored

# Parameters

- Formal parameters are defined in the method header
- Actual parameters are the values sent when the method is invoked
- When the method is invoked, it's as if there is assignment statement:

`formalParam = actualParam;`

# Method Example

method header

formal parameters- gone when  
the method ends

↓  
`public int add(int n1, int n2) {`

`int sum = n1 + n2;` sum is a local variable-  
`return sum;` gone when the method ends

`}`

return statement- sum is of type int

# Method Example

```
public static int add(int n1, int n2) {  
    int sum = n1 + n2;           formal parameters  
    return sum;                  (n1 and n2)  
}  
  
public static void main(String[] args) {  
    int num = 1;  
    int result = add(num, 2);  
}  
                                actual parameters  
                                (num and 2)
```

# Method Example (cont.)

- When add is invoked, it is as if we have the statements:

```
n1 = num;
```

```
n2 = 2;
```



# Methods and Garbage Collection

- When the method ends, all formal parameters and local variables are lost.

# Method Invocation

- To invoke a method in a different class:
  - invoking object (or class) followed by dot operator
  - method name
  - actual parameters

- Examples:

`d1.roll();`

`d1.setFaceValue(6);`

actual parameter

# Method Invocation

- If the method invoked is in the same class, you only need the method name.
  - You can omit the invoking object.
  - Or you can use `this` as the invoking object.

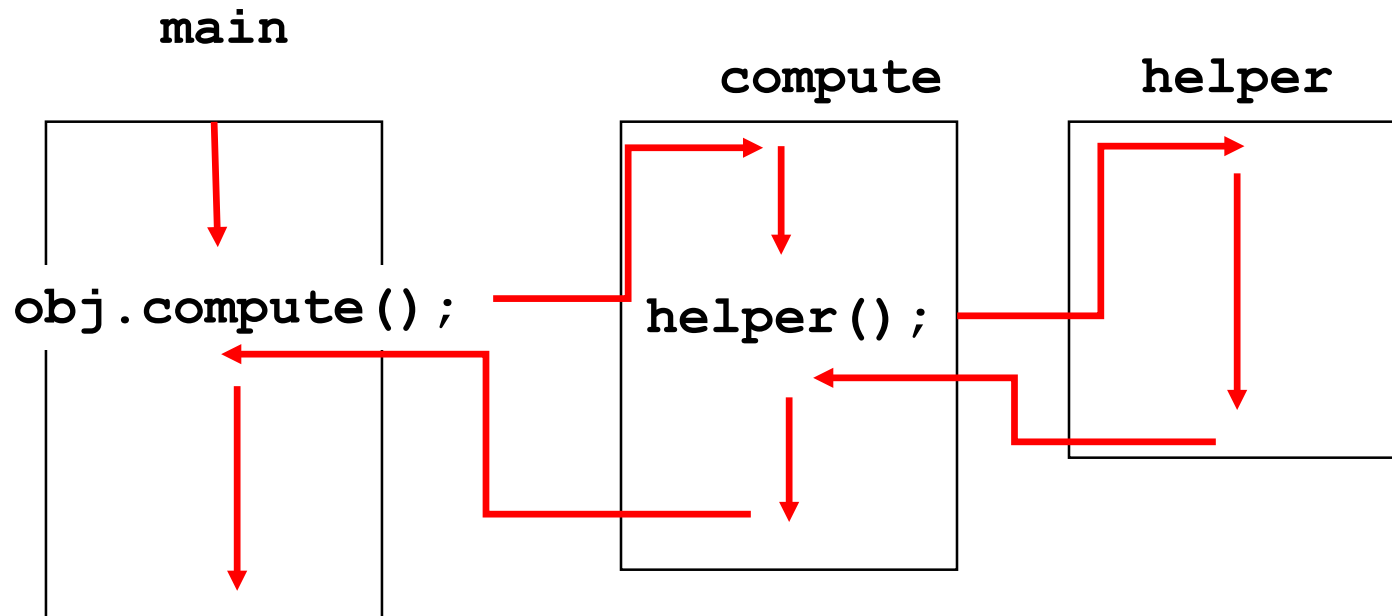
# Declaration vs. Invocation

- Declaring a method specifies what happens when it runs.
- Invoking a method actually *makes* it run.

# Flow of Control

- Java executes statements in order, starting with the first statement in the main method.
- When a statement contains a method invocation, the flow of control jumps to that method and executes that method line by line.
  - You can think of the original method as being *put on pause* until the invoked method completes.
- An invoked method is complete when you reach:
  1. a `return` statement or
  2. the end of the method
- When the method completes, control returns to where the method was called and continues.

# Flow of Control Example (cont.)



main method invokes the  
compute method with the  
obj object

# **PASS BY VALUE**

# What is Stored in Memory

- Primitive variables
  - The actual value- the data
- Object variables (also called object *references*)
  - A reference/ pointer/ memory address to the place in memory where all the information about the object resides
- This is a critical distinction in Java!



# Assignment Statements

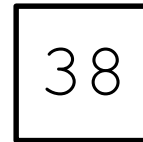
- Assignment takes the **value** on the right and stores it in the variable on the left.
- Think about what **the value** is!
  - It's different for primitives and objects!

# Assignment- Primitives

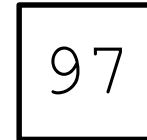
- Assignment takes the value on the right and stores it in the variable on the left.
  - For primitives, the value is just the data!

```
int num1 = 38;
```

```
int num2 = 97;
```



num1



num2

# Assignment- Primitives

```
num1 = num2;
```

- What is the **value** of num2?
- Because it's a primitive, the value is just the data!  
So the data- the actual number- is placed into num1.

97

num1

97

num2

# Assignment- Primitives

`num2++;`

97

`num1`

98

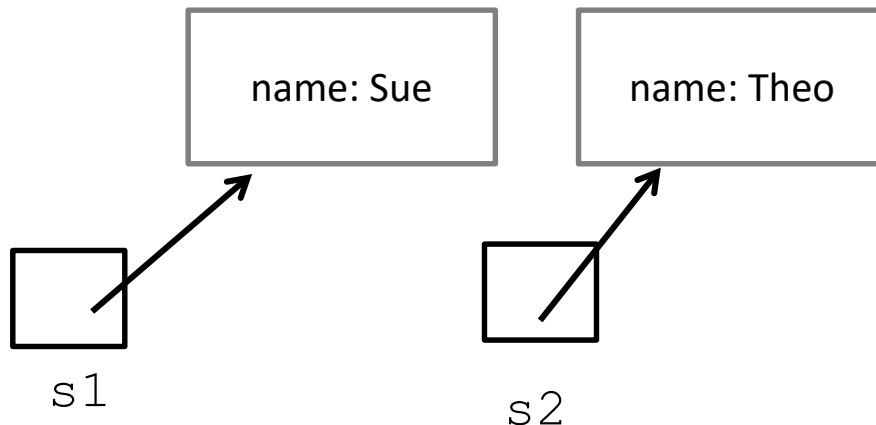
`num2`

# Assignment- Objects

- Assignment takes the value on the right and stores it in the variable on the left.
  - For objects, the value is a memory address!

```
Student s1 = new Student("Sue");
```

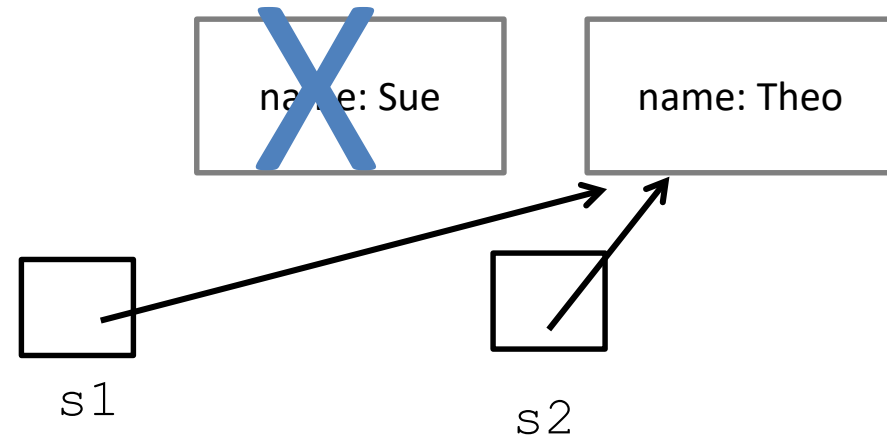
```
Student s2 = new Student("Theo");
```



# Assignment- Objects

```
s1 = s2;
```

- What is the **value** of s2?
- Because it's an object, the value is the address!
- So now s1 and s2 point to the exact same place in memory- the same address!
- Because no reference points to the other Student object, it gets garbage collected.

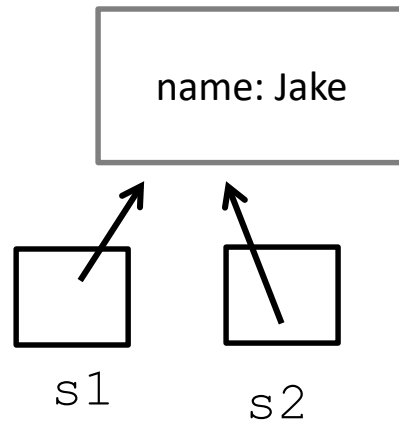


# Aliases

- s1 and s2 are now **aliases**.
- Variables that point to the same object (the same place in memory) are *aliases*
- Changing that object through one reference (i.e., one variable name) changes it for *all* references- because there is only **one** object!

# Aliases

```
s2.setName ("Jake") ;
```





# Invoking Methods with Parameters

- Formal parameters are defined in the method header
  - They last as long as the method lasts.
  - When the method is over, these parameters are gone!
- Actual parameters are the values sent when the method is invoked.

# Passing Parameters

- When a method is invoked, it's as if there is assignment statement executed behind the scenes:

```
formalParam = actualParam;
```

- This is an assignment statement!
  - When you use the assignment operator with objects, you create **aliases**.
  - Formal object parameters are **aliases** of actual parameters.

# Pass By Value

- Parameters in Java are ***passed by value***
- This means that the ***value*** of the actual parameter is *assigned to* the formal parameter.
  - But remember how assignment works for primitives vs objects!

# Objects as Parameters

- When an object is passed to a method, the actual parameter and the formal parameter become *aliases* of each other
  - If you change the internal state of the formal parameter by invoking a method, you change it for the actual parameter as well

# Review the PassingParameterExample

- In this example, we pass a primitive and an object into a method.

# Code Trace- Primitives

```
int num = 0;
```

num 0

# Code Trace- Primitives

```
primitiveParam(num) ;
```

num 0

```
// number = num
```

```
// value is assigned!
```

number 0

# Code Trace- Primitives

```
number = 99;
```

num 0

number 99



# Code Trace- Primitives

```
// method ends
```

num 0

```
// local variables and
```

```
formal parameters
```

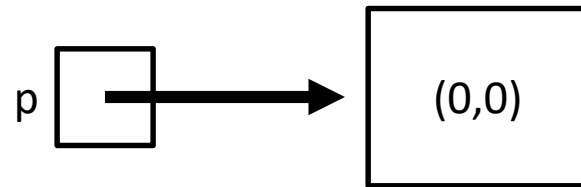
```
are garbage collected
```

number 99



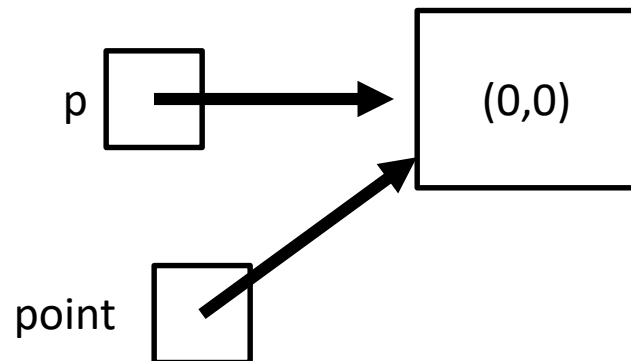
# Code Trace- Objects

```
Point p = new Point(0,0);
```



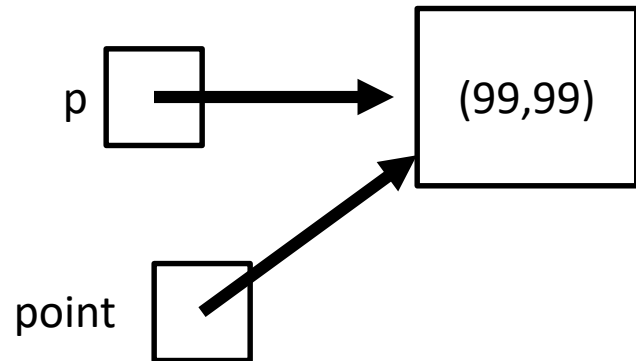
# Code Trace- Objects

```
objectParam(p) ;  
// point = p  
// value is assigned!  
// alias is created!
```



# Code Trace- Objects

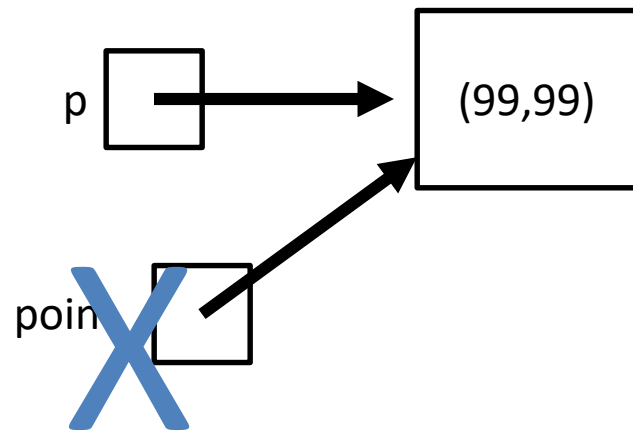
```
point.setLocation(99, 99);
```



# Code Trace- Objects

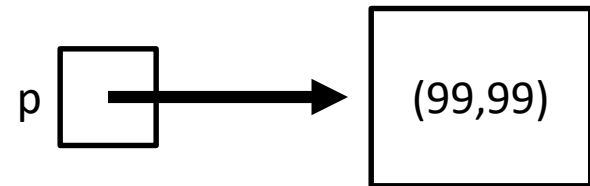
```
// method ends
```

```
// local variables and  
formal parameters  
are garbage collected
```



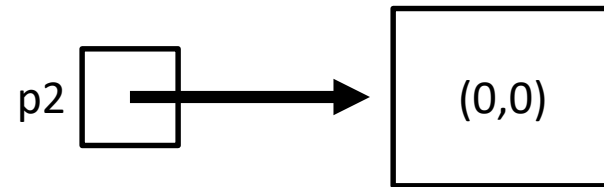
# Code Trace- Objects

```
// value is still changed back  
in main
```



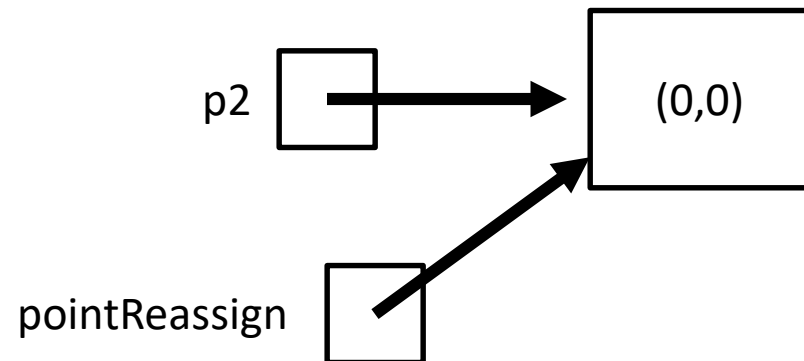
# Code Trace- Objects

```
Point p2 = new Point(0,0);
```



# Code Trace- Objects

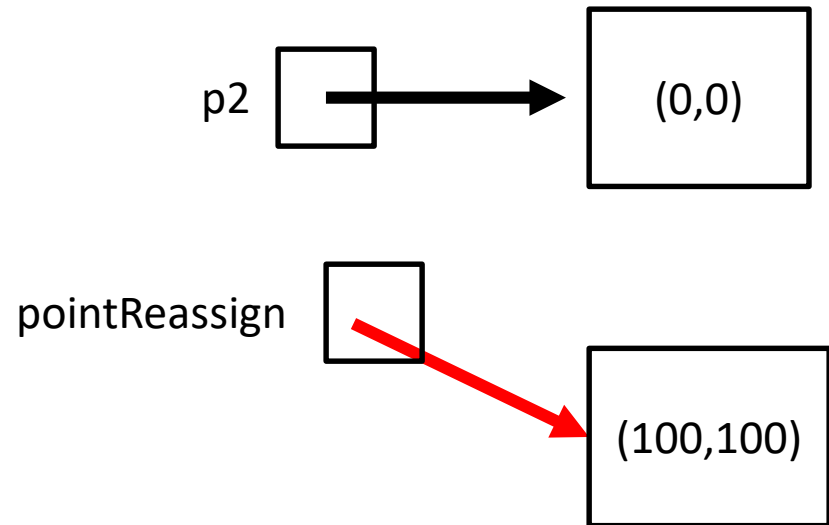
```
objectParamReassign(p2);  
// pointReassign = p2  
// value is assigned!  
// alias is created!
```





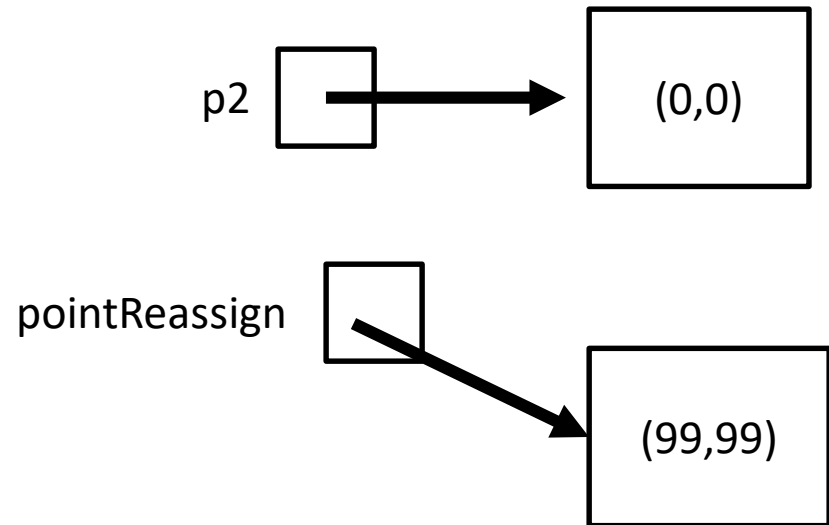
# Code Trace- Objects

```
pointReassign = new Point(100,100);  
// alias is broken!
```



# Code Trace- Objects

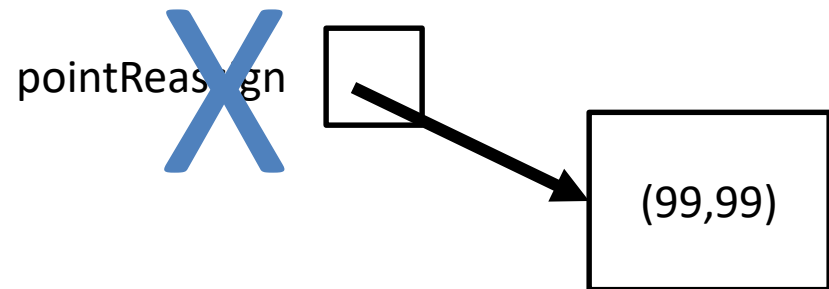
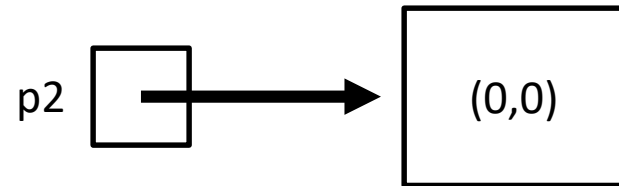
```
pointReassign.setLocation(99,99);
```



# Code Trace- Objects

```
// method ends
```

```
// local variables and formal  
parameters are garbage collected
```



# Key Points about Pass By Value

- Java is pass by value!
- The key is: what is the value?
  - For primitives: the actual data
  - For objects: the memory location/reference
- Using direct assignment with objects creates aliases. (**NOT** copies!)
- Passing objects to parameters creates aliases.
  - Invoking a method (with dot operator) inside the method changes the object inside and outside the method- because it's the same object!
  - Reassigning a formal parameter (formal parameter on left side of equal sign) breaks the alias link. This is usually a mistake.

# **WRITING CLASSES**

# Class Design

- A class is a *concept* of an object
- Once we define a class, we can instantiate as many objects of that class as we need
- Classes define a blueprint of what all objects of that class will look like.
  - The class is the blueprint. The object is the house.
  - The class is the recipe. The object is the cookie.

# Class Design

- Classes define:
  - Characteristics (state)
    - What describes the object
    - Represented by instance data
  - Functionality or actions (behaviors)
    - What the object can do or actions that can be done to the object
    - Represented by methods

# Class Building Blocks

- Instance data variables
- Constructor
- Getters and setters
- toString method
- Other class-specific methods



# Instance Data

- Represents the characteristics of the object
- Declared within a class but outside of methods
- Can be accessed by all methods within the class
- Each object has its **own version** of each instance data variable

# Constructor

- Create and set up an object
  - An object is an *instance of* the class
  - Initializes the instance data variables using values sent in as parameters or default values defined in the class as constants
- Called when an object is created with the keyword **new**
- Has the same name as the class
- Has **no** return type

# The **new** Keyword

- To create an object, use the **new** keyword and the constructor.
- This is called *instantiating the object* or *creating an instance of* the class.
- **new** does three things:
  - allocates the necessary memory for the object
  - executes the constructor
  - returns the address of (reference to) the object so it can be stored in the variable

# Creating Objects

```
Student s1 = new Student(51);
```

- **new** does three things:
  - allocates the necessary memory for the object
  - executes the code inside of the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



Student Class

id: ?

# Creating Objects

```
Student s1 = new Student(51);
```

- **new** does three things:
  - allocates the necessary memory for the object
  - **executes the code inside of the constructor**
  - returns the address of (reference to) the object so it can be stored in the variable

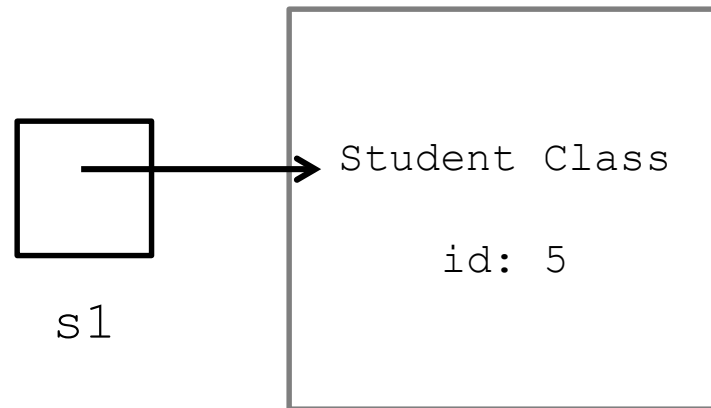
Student Class

id: 51

# Creating Objects

```
Student s1 = new Student(51);
```

- **new** does three things:
  - allocates the necessary memory for the object
  - executes the code inside of the constructor
  - returns the address of (reference to) the object so it can be stored in the variable



# Visibility Modifiers

- public
  - Can be referenced anywhere
- private
  - Can be referenced within a class
- protected
  - Related to inheritance... more to come later...
- default
  - can be referenced within the package... more to come later...

# Getters and Setters

- Provides access to the instance data
- Also called *accessors* and *mutators*
- Getter

- Returns the current value of a variable

```
public TYPE getVARNAME() {  
    return VARNAME;  
}
```

- Setter

- Updates the value of a variable
  - Can define parameters for valid values- validity checks

```
public void setVARNAME(TYPE NEWVALUE) {  
    VARNAME = NEWVALUE;  
}
```



# Validity Checks in Setters

```
public void setVARNAME(TYPE NEWVALUE) {  
    if( NEWVALUE meets some test) {  
        VARNAME = NEWVALUE;  
    } else {  
        // maybe do nothing  
        // maybe print an error message  
        // maybe throw an exception  
    }  
}
```

# toString Method

- Returns a text representation of the object
- Automatically called behind-the-scenes when the object is concatenated with a `String` or put inside a `println` method
- Has this **exact** header:

```
public String toString() {  
    return STRINGVAL;  
}
```

# Class Specific Methods

- The method header is used to declare the method:
  - Visibility
  - Return type
  - Method name
  - Formal parameters
    - List of type and name separated by commas
- Methods represent the functionality for the class.
- Methods often *change the state* of the object (i.e., update the value of the instance data variables).

# **INHERITANCE**

# Inheritance

- *Inheritance* allows you to design a new class from an existing class
  - The existing class is called the
    - *parent class*
    - *superclass*
    - *base class*
  - The new class is called the
    - *child class*
    - *Subclass*
    - *derived class*

# Inheritance

- Inheritance creates an *is-a* relationship.
  - The child *is a* more specific version of the parent.
- Examples:
  - Parent: BankAccount
  - Child: CheckingAccount, SavingsAccount
  
  - Parent: Student
  - Child: GraduateStudent, UndergraduateStudent

# Inheritance

- The child inherits characteristics of the parent
  - Methods
  - Data
- You can modify the child class by:
  - Adding new methods and variables
  - Modifying inherited methods
- Inheritance is established with the `extends` reserved word
  - Example: `public class GraduateStudent extends Student`

# Inheritance and Visibility

- private methods and variables in the parent class **are** inherited into the child class, but they **cannot** be directly accessed.
  - The child class has to use public getters and setters and public methods only, just like all other classes!
- Child classes *can* directly access protected methods and variables.
  - protected visibility does not provide as much encapsulation as private, so you should have a good reason to use it!



# Parent Class Constructors

- Constructors are *not* inherited, even though they have public visibility.
  - The child class does not inherit the constructor.
  - The child class needs its own constructor.
  - Often, we still need to set up the “parent’s part” of the object and do additional things for the “child’s part” of the object.
- It’s good practice to call the parent constructor and then add any additional set up needed in the child class.

# The `super` Reference

- Use the `super` reference to call the parent's constructor:
  - The child class's constructor calls the parent constructor.
  - The first line of the child constructor should use the `super` reference to call the parent's constructor.
  - Example: `super();` or `super(x, y, z);`
- Without an explicit call, the default constructor of the parent class will be automatically called: `super();`
  - If there is no default constructor in the parent class, you will get a compiler error.

# Overriding Methods

- A child class can *override* the definition of an inherited method and provide its own implementation.
- The new method has the same signature (name and parameters) as the parent's method, but will have a different implementation.

# @Override Annotation

- Annotations contain metadata about programs.
- You can use annotations to create new compiler checks.
- @Override will ensure at compile time that the method header is correct.
  - Always best to find an error at compile time rather than try to track it down at runtime!
- It's also a good visual cue to someone reading the code!
- It's good practice to always use this annotation when overriding a method!

# @Override Annotation

- Example:
  - In parent: `public void method() { ...}`
  - In child (mistake): `public void mthod() {...}`  
`// this is a mistake but will compile fine`
  - In child (mistake): `public void method(int n)`  
`{ ... } // this is a mistake but will compile fine`

# @Override Annotation

- Example:
  - In parent: `public void method() { ... }`
  - In child (mistake): `@Override public void mthod() { ... } // this will no longer compile`
  - In child (mistake): `@Override public void method(int n) { ... } // this will no longer compile`

# Overriding Methods

- When you override a method, you want to either:
  - Do something different
  - Do what the parent method does, but do more
- To *add on* to the parent's method, you can use the `super` reference.
  - Invoke the parent version of the method with `super.method()`;

# The `super` Reference

- Two uses of `super`:
  - Invoke the parent constructor (using `super(...)`)
    - Must be first line of the constructor!
  - Invoke the parent's version of an overridden method (using `super.parentMethod(...)`)
    - Can be used anywhere!
    - Can be used to invoke *any* parent method.



# **POLYMORPHISM**

# Object Type

- All variables in Java have a type.
  - Primitive or object
- Object variables actually have *two* types:
  - Declared Type
  - Actual Type

# Declared Type

- The declared type is the type listed when you first declare a variable.
  - Example:  
Student s;  
BankAccount b;  
String st;
- The declared type is declared only once and cannot be changed.

# Declared Type

- The compiler **only** knows about the declared type.
- In order for your code to compile, any method invoked must exist in the **declared** class.
  - Either directly or through inheritance.

# Actual Type

- Object variables also have an **actual type**.
- The actual type is based on how the object is created (usually by a call to the constructor).
- The actual type might change throughout the program or across different runnings of the program.

# Actual Type

- The actual type is not considered at compile time.
- The actual type is used at runtime.

# Actual Type and Declared Type

- The actual type must be **compatible with** the declared type.
- Actual class must be the same class, a child class, or any other class lower on the inheritance tree.
- If declared type is an interface, the actual class must be some class that implements that interface.

# Actual Type

- If the code compiles, we know that a method exists in the declared class.
- At run time, the actual type is used to determine **which version** of a method is invoked.
  - The most specific version will be invoked- the one closest to the actual type.



# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends  
Student {  
    ...  
    @Override  
    public void printInfo() {}  
}
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
}
```

```
Student student1; // declared type: Student  
                  // actual type not yet known- could be  
                  // Student or UndergradStudent
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
}
```

```
Student student1 = new Student();  
    // declared type: Student  
    // actual type: Student
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
}
```

```
Student student1 = new UndergradStudent();  
    // declared type: Student  
    // actual type: UndergradStudent  
    // allowed because UndergradStudent is child of Student
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
}
```

```
UndergradStudent student1 = new UndergradStudent();  
    // declared type: UndergradStudent  
    // actual type: UndergradStudent
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
}
```

```
UndergradStudent student1 = new Student();  
    // COMPILER ERROR!
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
  
}
```

```
Student student1 = new UndergradStudent();  
    // declared: Student; actual: UndergradStudent
```

```
student1 = new Student();  
    // declared: Student; actual: Student  
    // declared cannot change! actual might!
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}  
  
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
  
    public void earnBachelorsDegree() { }  
}  
  
Student student1 = new UndergradStudent();  
    // declared: Student; actual: UndergradStudent
```



# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}
```

```
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
  
    public void earnBachelorsDegree() { }  
}
```

```
Student student1 = new UndergradStudent();  
// declared: Student; actual: UndergradStudent
```

```
student1.printInfo();  
// allowed: printInfo exists in Student, the declared type  
// at runtime, the version from UndergradStudent is invoked, because that is the actual  
type
```

# Example

```
public class Student {  
    ...  
    public void printInfo() {}  
}  
  
public class UndergradStudent extends Student {  
    ...  
    @Override  
    public void printInfo() {}  
  
    public void earnBachelorsDegree() { }  
}  
  
Student student1 = new UndergradStudent();  
// declared: Student; actual: UndergradStudent  
  
student1.earnBachelorsDegree();  
// COMPILER ERROR!  
// earnBachelorsDegree() is not in the declared class
```

# Polymorphism

- *Polymorphism* means “having many forms”
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method invoked through a polymorphic reference can change from one invocation to the next
- In Java, all object references are potentially polymorphic

# Object References

- You can declare an object (left hand side) to be of a type high up on the inheritance hierarchy.
- You can then instantiate that object (right hand side) to be that same type or any type lower on the hierarchy.

# Example

```
public class Employee
public class UnpaidEmployee extends Employee
public class PaidEmployee extends Employee
public class SalariedPaidEmployee extends PaidEmployee
public class HourlyPaidEmployee extends PaidEmployee
```

# Example


```
public class Employee
public class UnpaidEmployee extends Employee
public class PaidEmployee extends Employee
public class SalariedPaidEmployee extends PaidEmployee
public class HourlyPaidEmployee extends PaidEmployee
```

```
Employee e;
    // declared type: Employee
    // actual type could be any of these 5 classes!
```


# Example

```
public class Employee
public class UnpaidEmployee extends Employee
public class PaidEmployee extends Employee
public class SalariedPaidEmployee extends PaidEmployee
public class HourlyPaidEmployee extends PaidEmployee
```

```
Employee e = new Employee();
Employee e = new HourlyPaidEmployee();
PaidEmployee e = new SalariedPaidEmployee();
PaidEmployee e = new UnpaidEmployee(); // COMPILER ERROR
```



declared type



actual type

# Using Polymorphism

- Why would we want to use a different declared and actual type?
- We often don't with a single variable.
- Polymorphism allows us to group together objects into one collection!



# Example

```
public class Employee
public class UnpaidEmployee extends Employee
public class PaidEmployee extends Employee
public class SalariedPaidEmployee extends PaidEmployee
public class HourlyPaidEmployee extends PaidEmployee

ArrayList<Employee> list = new ArrayList<Employee>();
list.add(new Employee());
list.add(new HourlyPaidEmployee());
list.add(new SalariedPaidEmployee());
list.add(new UnpaidEmployee());
```

# Example

```
public class Employee
public class UnpaidEmployee extends Employee
public class PaidEmployee extends Employee
public class SalariedPaidEmployee extends PaidEmployee
public class HourlyPaidEmployee extends PaidEmployee
```

```
ArrayList<Employee> list = new ArrayList<Employee>();
    // all objects in the list have declared type Employee
list.add(new Employee()); // actual type: Employee
list.add(new HourlyPaidEmployee()); // actual: Hourly
list.add(new SalariedPaidEmployee()); // actual: Salaried
list.add(new UnpaidEmployee()); // actual: Unpaid
```

# Example

```
public class Employee {
    public void hire() {}
}
public class UnpaidEmployee extends Employee {
    @Override
    public void hire() { }
}
public class PaidEmployee extends Employee {
    public void pay() {}
}
public class SalariedPaidEmployee extends PaidEmployee {
    @Override
    public void pay() {}
}
public class HourlyPaidEmployee extends PaidEmployee

ArrayList<Employee> list = new ArrayList<Employee>();
list.add(new Employee()); // invoke version in Employee
list.add(new HourlyPaidEmployee()); // invoke version in Employee
list.add(new SalariedPaidEmployee()); // invoke version in Employee
list.add(new UnpaidEmployee()); // invoke version in UnpaidEmployee
for(Employee e : list) {
    e.hire();
    // compiles because there is a hire() method in the Employee class
    // will invoke the proper version based on runtime, actual type
}
```

# Example

```
public class Employee {
    public void hire() {}
}
public class UnpaidEmployee extends Employee {
    @Override
    public void hire() { }
}
public class PaidEmployee extends Employee {
    public void pay() {}
}
public class SalariedPaidEmployee extends PaidEmployee {
    @Override
    public void pay() {}
}
public class HourlyPaidEmployee extends PaidEmployee

ArrayList<Employee> list = new ArrayList<Employee>();
list.add(new Employee());
list.add(new HourlyPaidEmployee());
list.add(new SalariedPaidEmployee());
list.add(new UnpaidEmployee());
for(Employee e : list) {
    e.pay();
    // COMPILER ERROR- pay does not exist in declared class Employee
}
```

# Example

```
public class Employee {
    public void hire() {}
}
public class UnpaidEmployee extends Employee {
    @Override
    public void hire() { }
}
public class PaidEmployee extends Employee {
    public void pay() {}
}
public class SalariedPaidEmployee extends PaidEmployee {
    @Override
    public void pay() {}
}
public class HourlyPaidEmployee extends PaidEmployee

ArrayList< PaidEmployee > list = new ArrayList< PaidEmployee >();
list.add(new Employee()); // COMPILER ERROR
list.add(new HourlyPaidEmployee());
list.add(new SalariedPaidEmployee());
list.add(new UnpaidEmployee()); // COMPILER ERROR
for(PaidEmployee e : list) {
    e.pay();
}
```

# Example

```
public class Employee {
    public void hire() {}
}
public class UnpaidEmployee extends Employee {
    @Override
    public void hire() { }
}
public class PaidEmployee extends Employee {
    public void pay() {}
}
public class SalariedPaidEmployee extends PaidEmployee {
    @Override
    public void pay() {}
}
public class HourlyPaidEmployee extends PaidEmployee

ArrayList< PaidEmployee > list = new ArrayList< PaidEmployee >();
list.add(new Employee());
list.add(new HourlyPaidEmployee()); // invokes version in PaidEmployee
list.add(new SalariedPaidEmployee()); // invokes version in SalariedPaidEmployee
list.add(new UnpaidEmployee());
for(PaidEmployee e : list) {
    e.pay();
    // compiles because there is a hire() method in the PaidEmployee class
    // will invoke the proper version based on runtime, actual type
}
```

# Polymorphism

- This is polymorphism!
- An employee object invoking the pay method could invoke different versions of the method depending on the *actual type* of the object.

# Summary

- You can declare an object (left hand side of assignment statement) to be of a type high up on the inheritance hierarchy. You can then instantiate that object (right hand side of assignment statement) to be of any type lower on the hierarchy.



# Summary

- When you declare an object variable, you specify the declared type (left hand side of the assignment).
  - The declared type cannot change.
  - The compiler only knows about this type.
  - The compiler only allows you to invoke methods in this declared class.
- At runtime, the JVM knows the *actual* type of the object (right hand side of the assignment).
  - It is based on how the object was created (e.g., a constructor).
  - It could be the declared type or any subclass of the declared type.
  - The actual type might not always be the same at different times in the same program or at different runnings of the program.
  - The actual type is used to invoke the correct version of a method.

# Downcasting

- You *can* invoke a method that exists in the child class actual class but not the parent class declared class.
- To do this, you have to *downcast*.
- You should always use the `instanceof` operator with a downcast.

# Example

```
public class Employee {  
    public void hire() {}  
}  
public class PaidEmployee extends Employee {  
    public void pay() {}  
}
```

```
ArrayList<Employee> list = new ArrayList<Employee>();  
list.add(new Employee());  
list.add(new HourlyPaidEmployee());  
list.add(new SalariedPaidEmployee());  
list.add(new UnpaidEmployee());  
for(Employee e : list) {  
    e.pay(); // COMPILER ERROR- pay does not exist in Employee  
}
```

# Example

```
public class Employee {  
    public void hire() {}  
}  
public class PaidEmployee extends Employee {  
    public void pay() {}  
}
```


```
ArrayList<Employee> list = new ArrayList<Employee>();  
list.add(new Employee());  
list.add(new HourlyPaidEmployee());  
list.add(new SalariedPaidEmployee());  
list.add(new UnpaidEmployee());  
for(Employee e : list) {  
    if(e instanceof PaidEmployee) {  
        PaidEmployee p = (PaidEmployee) e;  
        p.pay();  
    }  
}
```

# Example

```
public class Employee {  
    public void hire() {}  
}  
public class PaidEmployee extends Employee {  
    public void pay() {}  
}
```

```
ArrayList<Employee> list = new ArrayList<Employee>();  
list.add(new Employee());  
list.add(new HourlyPaidEmployee());  
list.add(new SalariedPaidEmployee());  
list.add(new UnpaidEmployee());  
for(Employee e : list) {  
    if(e instanceof PaidEmployee) {  
        ((PaidEmployee) e).pay();  
    }  
}
```

tells the compiler  
to temporarily  
treat e like a  
PaidEmployee; does  
**not** change the type  
of e!



# Downcasting

- instanceof is a safety check that you have the right type.
  - Best practice!!!
- The cast is the command to the compiler.
- You need to do both!

# Example

```
public class Employee {  
    public void hire() {}  
}  
public class PaidEmployee extends Employee {  
    public void pay() {}  
}
```

```
ArrayList<Employee> list = new ArrayList<Employee>();  
list.add(new Employee());  
list.add(new HourlyPaidEmployee());  
list.add(new SalariedPaidEmployee());  
list.add(new UnpaidEmployee());  
for(Employee e : list) {  
    if(e instanceof PaidEmployee) {  
        e.pay(); // COMPILER ERROR- declared type is still  
                // Employee because the compiler evaluates this  
                // line on its own- it doesn't look at the if  
    }  
}
```

# Example

```
public class Employee {
    public void hire() {}
}

public class PaidEmployee extends Employee {
    public void pay() {}
}
```

```
ArrayList<Employee> list = new ArrayList<Employee>();
list.add(new Employee());
list.add(new HourlyPaidEmployee());
list.add(new SalariedPaidEmployee());
list.add(new UnpaidEmployee());
for(Employee e : list) {
    ((PaidEmployee) e).pay(); // will compile, but this is
    // dangerous because if the object isn't (actual)
    // type PaidEmployee, the code will throw a
    // runtime exception and crash
}
```



# **THE EQUALS METHOD**

# The Object Class

- The `Object` class is the ultimate root of all class hierarchies
- `Object` is defined in the `java.lang` package
- *All* classes in Java are derived from `Object`
- If a class does not explicitly extend a class, it is automatically a child of the `Object` class

# The Object Class Methods

- There are a few useful methods in `Object` that are inherited by all classes:
- `public String toString()`
  - Every time you write your own `toString` method, you are actually overriding this method from the `Object` class.
  - This is why the method header must match exactly!
  - In the `Object` class, this method returns a `String` with the name of the object's class and some other information (the memory address).
  - You should override this method with a text representation of your object.

# The Object Class Methods

- `public boolean equals (Object obj)`
  - The inherited version returns true if two references are aliases
  - You can override this method to define *meaningful* or *logical* equality in a more appropriate way
    - Examples: same ID, same name, etc.
  - Example: `String` overrides `equals` to return true if two `String` objects contain the same characters

# Implementing Equals

- We need to first check that we've been given an object with the right type.
  - We do this with the `instanceof` operator
  - `instanceof` returns a boolean that represents whether an object is an *instance of* a class
- If the parameter is the right type, we use a *cast* to tell the compiler that we want to treat the parameter as that type.
- Then we compare whatever information we are using for equality.
  - Will often use the `equals` method (or `equalsIgnoreCase` method) to compare String variables and `==` to compare numeric variables.

# Implementing Equals

```
@Override
public boolean equals(Object obj) {
    if(obj instanceof MyClass) {
        MyClass otherObject = (MyClass) obj;
        // compare whether the variables are the same
    } else {
        return false;
    }
}
```

# The equals Method header

- A common error is to accept a parameter of your class, instead of object.
- This is incorrect because this method no longer overrides the inherited method!
- Be very careful not to do this!

```
public boolean equals(Employee e)
```

<http://tinyurl.com/equalsMethodHeader>

**NULL**



# null

- null is a keyword/reserved word in Java
- null represents **no value**

# null as a Default Value

- Instance data variables are given a default value when they are declared.
  - Local variables are not.
- Each data type has a value that is used for the default value.
- null is the default value for objects when they have been declared but not initialized.
  - `private int age; // default value for int is 0`
  - `private String name; // default value of name is null`
  - `private Student s; // default value of s is null`

# null as a Value

- null is a value that can be assigned to any *object* reference (variable).
- null cannot be assigned to primitives.
- Examples:
  - Student s = null; // allowed
  - int n = null; // not allowed
  - Integer m = null; // allowed

# What can you do with null?

- Compare it with ==
- Examples:
  - `if(student!=null) { ... }`
  - `if(student!=null && student.meetsCriteria()) { ... }`

# What **can't** you do with null?

- Pretty much anything else!
- Most importantly: you cannot invoke any methods or try to access any variables on null.
- Invoking a method on a null object will throw a `NullPointerException`, which will crash your program.
  - This is a bad exception because it is almost always a result of programmer error.
  - This is almost always entirely preventable.

# What **can't** you do with null?

- Example:
  - `String s = null`
  - `s.equals("hello");` // crash!
  - `s=="hello";` // allowed- will not crash, returns false
- Example:
  - `Student student = null;`
  - `System.out.println(student.name);` // crash!

# null and Linked Nodes

- null will matter a lot when we cover nodes and linked lists!!

# null and Strings

- A string whose value is the empty String is **not** null.
- An empty String is a String that does not contain any characters. (But it is not null!)
  - `String s1 = "";` // empty string! length = 0
  - `String s2 = null;` // no length- no value!
  - `s1==s2;` // false



# null and Strings

```
String s1 = "";  
String s2 = null;  
String s3; // value is also null
```

```
System.out.println(s1.toUpperCase());  
// allowed- but nothing will be printed!
```

```
System.out.println(s1.length());  
// allowed and will print 0
```

```
System.out.println(s2.toUpperCase());  
// not allowed- will crash with NullPointerException
```

```
System.out.println(s3.length());  
// not allowed- will crash with NullPointerException
```

# **CODING CONVENTIONS**

# Conventions- Why do they matter?

- Makes code more readable
- Makes code easier to maintain (by you or, more likely, by others)
- Provides a quick signifier to the reader of the type of information (e.g., variable, constant, class)
- Offers an indicator of the quality of your code
  - Oracle: If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.
- More in-depth discussion:  
<http://hilton.org.uk/presentations/naming-guidelines>

# Java Conventions

- Oracle:

<https://www.oracle.com/technetwork/java/javase/overview/codeconvtoc-136057.html>

- Google: <https://google.github.io/styleguide/javaguide.html>

- GitHub:

<https://github.com/twitter/commons/blob/master/src/java/com/twitter/common/styleguide.md>

# Variables

- lowerCamelCase
  - first letter lowercase, first letter of each internal word capitalized
- descriptive and meaningful
  - avoid one-character names except for temporary variables (like looping variables)
- This applies to parameters, too!

# Constants

- `ALL_CAPS_WITH_UNDERSCORES`
- From Oracle: Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.

# Methods

- lowerCamelCase
- descriptive
- often contains a verb

# Classes and Interfaces

- UpperCamelCase
  - first letter and each internal word capitalized
- Simple and Descriptive
- Often a singular noun
- Use whole words
- Avoid acronyms and abbreviations (unless abbreviation is much more widely used, such as URL or HTML)



# Others

- Oracle: It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others- you shouldn't assume that other programmers know precedence as well as you do.
- Use indentation and whitespace to improve readability

# A Final Thought

“Coding conventions should be used as a guide, not as a sacred, immutable law of nature.

Those with less experience will initially benefit from following conventions, but as you gain experience, you should rely more on your own sense of taste, as opposed to blindly following the rules. The problem with rules is that there are many cases in which they should be broken.

In general, the question in your mind should always be one of **compassion for the reader**: *"Will this help or hinder a maintainer who is unfamiliar with my code?"*. That is the guiding principle.”

Source: <http://www.javapractices.com/topic/TopicAction.do?Id=115>