Super-Classes and sub-classes

- Subclasses.
- Overriding Methods
- Subclass Constructors
- Inheritance Hierarchies
- Polymorphism
- Casting

<u>Subclasses</u>:

- Often you want to write a class that is a special case of an existing class.
- For example, you have an employee class and you want to have a special class for employees who are salespeople.
- Salespeople are employees, so they share all the qualities of employees, but they also make commission which other employees don't
- We don't want to rewrite all the employee information for the new class, so we make Salesperson a *subclass* of Employee.
- Informally, a subclass satisfies the "is-a" criteria. A salesperson "is a" employee.

<u>Subclasses</u>:

- A subclass *inherits* fields and methods from it's *parent*.
- Other names for the parent of a subclass are *superclass* and *base class*.
- Other names for the sub-class of a parent are *derived class* and *child class*.
- The *hierarchy* (sub-class/superclass relationships) of classes can go many levels.
- A class can have exactly one parent class, but it might have many child classes.

Subclasses:

- Back to our example, suppose the Employee class starts like:

```
public class Employee {
  private String name;
  private double salary;

  public Employee(String newName, double newSalary) {
    name = newname;
    salary = newSalary;
  }
 . . .
```

- The Salesperson class also needs a name and salary, but it doesn't need to define them, since it inherits them from its super class.

Subclasses:

- We define the Salesperson class like this:

```
public class Salesperson extends Employee {
  private double salesAmt;
  private double rate;
 . . .
  public double getRate() {
    return rate;
  }
. . .
```

- Notice the keyword **`extends`** which tells the compiler that Salesperson is a subclass of Employee.
- We don't need to declare **`name`** and **`salary`** in the Salesperson class because it inherits them from its parent.

Subclasses:

- Suppose now we have a reference to a Salesperson object:
  ```
  SalesPerson p = . . .
  ```
- We can call any method defined in salesperson:
  ```
  double r = p.getRate();   // Salesperson method returns rate
  ```
- Or any method from the parent class:
  ```
  String name = p.getName(); // Employee method
  ```
- This does not work the other way. An Employee object can't call methods defined only in Salesperson.
  ```
  Employee e = . . .
  double r = e.getRate(); // generates a compiler error
  ```

Overriding Methods:

- What if we want a method in the child class to behave differently than in it's parent class.
  - For example, suppose the Employee class contained the following method:
    ```
    double getSalary() {
      return salary;
    }
    ```
  - But for a salesperson we want the **`getSalary`** method to include the commission (the **`salesAmt * rate`**).
  - We can write a new **`getSalary`** method in our **`SalesPerson`** class.
  - When a **`SalesPerson`** object calls the **`getSalary`** method, the method defined in the **`SalesPerson`** class will be called instead of the one in the **`Employee`** class.
- This is called *overriding* the method of the superclass.

Overriding Methods:

- So how can we write the new getSalary method? The following won't work:

```
double getSalary() {
  return salary + salesAmt * rate;
}
```

  - This is because the **salary** field is private to the Employee class, so the SalesPerson class does not have access to it.
  - How can we get around this?
  - Call the accessor method?
  - This will also NOT work:

```
double getSalary() {
  return getSalary() + salesAmt * rate;
}
```

  - The problem is **getSalary()** will call the method in this class, which is itself!

Overriding Methods:

- The solution is to use the *super* keyword:

```
double getSalary() {
  return super.getSalary() + salesAmt * rate;
}
```

  - super refers to a class's parent, so in this case **`super.getSalary()`** calls the **`getSalary`** method from the **`Employee`** class.

Subclass Constructors:

- Here is a constructor for our **SalesPerson** class:

```
public SalesPerson(String newName, double newSalary,
                   double newRate, double amt) {
  super(newName, newSalary);
  salesAmt = amt;
  rate = newRate
}
```

  - Here **super** is used to call the constructor of the parent class.
  - This is needed since the **SalesPerson** class does not have access to private fields in the **Employee** parent class.
  - The call using super must be the first statement in the constructor.
  - If the call to the parent constructor is missing, the compiler will add one that calls the parent constructor with no arguments.
  - This will cause an error if the parent class does not have a constructor with no arguments defined.

- You may assign a child object to a reference to its parent. For example:

```
Employee emp = new SalesPerson("Paul", 200000, .2, 1050);
```

  - This is legal is a SalesPerson "is an" Employee.
- The reverse is NOT legal. You can't assign a parent object to a child reference:

```
SalesPerson sp = new Employee("John",  200000); //error!
```

  - The statement above is cause a compiler error since an Employee is not necessarily a SalesPerson.
- In the example on top, even though emp actually points to a SalesPerson, the reference is of type Employee, so you can't use it to access SalesPerson specific methods.

```
double r = emp.getRate(); // ERROR!
```

  - This will cause a compiler error because the **`Employee`** class does not have a method called **`getRate`**.

```
Employee emp = new SalesPerson("Paul", 200000, .2, 1050);
```

- The emp variable can call any Employee methods:

```
String name = emp.getName();  // legal
```

- What if you call a method in the parent that has been overridden in the child?
- For example suppose we make the call:

```
double salary = emp.getSalary();
```

  - Which version of getSalary is called?
  - The version defined in the SalesPerson class.

- The fact an object variable can refer to objects of different types is called *polymorphism*.
- One way this is very useful is that I can write a method that acts on a parent class, and can send it any object that is descended from that class.
- For example I could write a method to print paychecks:

```
public void printPaycheck(Employee emp) {
  . . .
```

  - And I can call it using a **`SalesPerson`** object as an argument.
  - I do NOT have to write a different method for every type of employee.

Casting:

- Just like you can use a cast to force type conversions where you might lose information:

```
double d = 45.6;
float f = (float) d;
```

- You can also use a cast to tell the compiler a class reference is really to an inherited class type.

```
SalesPerson sp = new SalesPerson(...);

Employee emp = sp; // legal because a SalesPerson is an Employee

SalesPerson sp2 = emp; // Compile ERROR!!
```

  - The last statement will generate a compiler error. Not all Employees are SalesPersons
  - However if I know the object referred to by emp is a SalesPerson I can use a cast.

```
SalesPerson sp = (SalesPerson) emp;  // legal
```

Casting:

- You must be careful using casting. Casting a reference to on object down in the inheritance chain will avoid a compiler error.
- However, you will have a runtime exception if the object is not of the type you cast.

```
Employee emp = new Employee();
SalesPerson sp = (SalesPerson) emp; // Runtime ERROR!!
```

  - The last statement will compile, but cause an error at runtime.
- Note that you can't use a cast to try to cast unrelated types.

```
String str = "I'm a salesperson!";
SalesPerson sp = (SalesPerson) str; // Compiler ERROR!!
```

  - The last statement will fail at compile time. The compiler knows the classes are unrelated.

Casting:

- Why would anyone ever need to do casting?
- Suppose we had our general printCheck method that printed the checks of all employees.

```
public static void printCheck(Employee emp) {
```

  - As we saw we can send this method a reference to a `SalesPerson` or an `Employee` object
  - This is great because it will do the same thing for both.
  - But what if we wanted to print a gold star on the paychecks of salespeople whose sales amount was above 10000?
  - Inside the method could we write?

```
if (emp.getSalesAmt() > 10000) {
```

  - No, because the Employee class does not have a **getSalesAmt** method!

Casting:

- We can use a cast.

```
public static void printCheck(Employee emp) {
    . . . // code to write most of check
    SalesPerson sp = (SalesPerson) emp;
    if (sp.getSalesAmt() > 10000) {
        . . .
```

  - This will compile. ☺
  - It will work fine when a **SalesPerson** object is sent to the method ☺
  - It will break at runtime if the object is not a **SalesPerson** method. ☹
  - How can we tell at runtime if casing is safe?
  - You can use the **instanceof** operator to check if the type is correct:

```
if (emp instanceof SalesPerson) {
  SalesPerson sp = (Salesperson) emp;
      . . .
```

Abstract Classes:

- Sometimes you don't want to implement all the methods for a class that's going to be used as a super class.
- Take our **`Employee`** class example from before. Say a store wants to have classes for employees, contractors, customers, and suppliers.
  - Perhaps the programmer decides to create a parent class called **`Person`** for all of these.
  - This class would contain fields common to all, like perhaps name, address, phone number, etc.
  - It might even contain common methods like **`printEnvelope()`**
  - However, imagine a method for granting access to a room which requires us to know what type of person (employee, customer, etc.) the object is.
  - I want to be able to say every **`Person`** has such a method, but I can't define it for a person in general.

Abstract Classes:

- A method can be declared as **abstract**. For example:
  `public abstract boolean roomAccess();`
  - Notice there is no implementation.
- An abstract method is not implemented in the class it is declared in.
- Abstract methods act as placeholders for methods that are implemented in the subclasses.
  - In this case the **`roomAccess()`** method should be implemented in the **`Employee`**, **`Customer`**, **`Supplier`**, etc classes
- If a class has one or more abstract methods, then it must be declared to be abstract.

```
public abstract class Person {
    . . .
    public abstract boolean roomAccess();
    . . .
```

Abstract Classes:

- An abstract class cannot be instantiated.
- In our previous example Person is an abstract class.
- You may have a Person reference variable:
  `Person her;  // legal and good`
- But you can't create a Person object:
  `her = new Person();  // compiler ERROR`
- So how can I even use a **`Person`** reference?
- I can use it to refer to any object of a subclass of Person.

```
her = new Employee();
```

Abstract Classes:

- Just because abstract classes cannot be instantiated, does not mean they can't have constructors.
- For example:

```
public abstract class Person {
  private String name;
    . . .
  public Person(String newName) {
    name = newName;
  }
    . . .
```

- Why would I want to have constructors defined if I can't have statements that include **`new Person`**?

Protected Access:

- We have talked about and mostly used the **`public`** and **`private`** modifiers.
- We now know enough to understand all the modifiers:

1. Private – Visible to the class only.
2. Public – Visible to the world.
3. Protected – Visible to subclasses and to the package.
4. Default (no modifier) – Visible to the package.

- The recommendation is that you mostly use public or private modifiers.
- All fields should be made private (to support encapsulation)

The Object Class:

- The *Object* class is at the top of the java class hierarchy.
- Every class in Java is a descendent of the Object class.
- You don't ever write something like:

```
public class MyClass extends Object
```

- Any class defined without an extends keyword is automatically a child of the Object class.
- This means a variable of type Object can reference any object.

```
Object obj = new AnyClass(); // legal
obj = "I'm now a string."; // legal since String is an object
obj = new int[34]; // also legal
```

<u>The Object Class</u>:

- Do you think the Object class contains any fields?
- The Object class has no fields, but several methods.
- Even though some of the methods don't really do anything if they are not overridden, the Object class is not abstract.
- You can find a description of the Object class on the Java API: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

The equals Method Revisited:

- **`equals(Object obj)`** is a method of the Object class.
- By default the **`equals`** method does the same thing as **`==`**
- In other words **`obj1.equals(obj2)`** is true iff **`obj1`** and **`obj2`** refer to the same object (same location in memory).
- Often we choose to override this method.

<u>The equals Method Revisited</u>:

- The Java Language Specification requires that an equals method meets the following:

1. It is *reflexive* (**`x.equals(x)`** is **`true`**)

2. It is *symmetric* (**`x.equals(y)`** iff **`y.equals(x)`**)

3. It is transitive (**`x.equals(y) && y.equals(z) => x.equals(z)`**)

4. It is *consistent* (**`x.equals(y)`** should return the same value every time called if x and y have not changed.)

5. **`x.equals(null)`** should be **`false`**.

The equals Method Revisited:

- We had earlier slides where we defined an equals method like:

```
public boolean equals(Employee otherEmp) {
   . . .
```

  - As part of the definition of the **`Employee`** class.
  - Does this override the **`Object`** class **`equals`** method?
  - No, because the parameter is not an **`Object`**, this overloads the method, but does not override the existing method.
- To override the Object method equals we need to write

```
public boolean equals(Object otherObj) {
  . . .
```

- But now we will need to be able to know what type of object is being sent as an argument.

The equals Method Revisited:

```
 public boolean equals(Object otherObj) {
   . . .
```

- To tell whether the object being tested is the same type as the class the method is defined in, we might be able to use the **`instanceof`** operator.

```
public class Employee {
    . . .
  public boolan equals(Object otherObj) {
    . . .
    if (!(otherObj instanceof Employee))
      return false;
    . . .
```

- The expression **`a instanceof b`** returns true iff **a** is the same class or **b** descendant of **b**
- This works great if we don't want to override the equals method in our decedents, but it can cause transitivity problems otherwise.

The equals Method Revisited:

- If you want to get the class of an object, you can use the **`Object`** method **`getClass()`**.
- You can use this in your **`equals`** method to check whether the objects are of the same type. e.g.

```
public boolean equals(Object otherObj) {
  . . .
  if (getClass() != otherObj.getClass())
    return false;
  . . .
```

- Use this if you don't want to have parent and child classes to be considered equal

The equals Method Revisited:

- Cay Horstmann in **Core Java** recommends the following formula for writing **`equals`** methods:

```
public boolean equals(Object otherObj) {
```

1. Test whether the objects refer to same place:
   `if (this == otherObj) return true;`
2. Test whether the other object is null:
   if (otherObj == null) return false;
3. Compare the classes using **`instanceof`** or **`getClass()`** depending on your definitions:
4. Cast the other object to your class type
5. Compare all the fields of the class, depending on your definition of equals, using the equals method for objects.

The hashCode Method:

- The **`Object`** class has a **`hashCode()`** method which returns an integer.
- This is actually the hash function that the HashMap class uses.
- Technically, if you override the **`equals`** method of a class, you should override the **`hashCode`** method as well.
  - This is because according to the Java Language spec if two objects are equal according to the **`equals`** method, then the should produce the same hash code.
- We won't worry about that in this class, but is something you should be aware of as you become Java programmers.

The toString Method:

- We've talked about the `toString()` method before.
- This is actually a method of `Object` class, which is why things like print can use it automatically.
- The default implementation of this method is not very informative. Try printing an object that doesn't have this method overridden some time to see what it looks like.

Interfaces:

- An **interface** is a set of requirements for a class.
- An interface gives a list of methods that the class must implement to satisfy the interface.
- For example:

```
public interface Comparable {
  int compareTo(Object other);
}
```

- This says any class that implements the **`Comparable`** interface must have a **`compareTo`** method.
- Notice in the interface definition there is not access level on the method. All methods in an interface are automatically public.

Interfaces:

- Why do we want interfaces?
- Remember the QuickSort class/method you wrote.
  - It sorted an array of integers
  - The same logic could have sorted and array of Strings or Doubles or anything else that has a notion of greater than.
  - The only thing that would change is how you compare the items.
  - The **`compareTo`** method gives a way to compare objects.
  - If you know a class implements the Comparable interface, then you know it has a compareTo method you can call and you can sort the items.
- The **`Arrays`** class has a **`sort`** method that works for any array of objects that implement the **`Comparable`** interface.

Interfaces:

- Suppose I want to be able to sort arrays of Employees
- Then I would have to have the **`Employee`** class implement the **`Comparable`** interface.
- I indicate this in the header for the class using the **`implements`** keyword.

```
public class Employee implements Comparable {
```

- I also have to include a **`compareTo`** method.

```
  public int compareTo(Object otherObj) {
    Employee other = (Employee) otherObj;
    return Long.compare(idNum, other.idNum);
  }
```

  - Here we are ordering by idNum, but we might choose by name or salary instead.

Interfaces:

- You can actually implement the generic Comparable interface which saves you from having to do a cast in your method:

```
public class Employee implements Comparable<Employee> {

  . . .

  public int compareTo(Employee other) {
    return Long.compare(idNum, other.idNum);
  }
```

Interfaces Properties:

- An interface is NOT a class. It can not be instantiated.
- For example the following gives an error:

```
c = new Comparable( . . .); // Compiler ERROR
```

- You can, however, have reference variables with interface types:

```
Comparable c = new Employee("Cindy", 75000); // correct
```

- You can also use the **instanceof** operator to check if an object implements an interface:

```
if (c instanceof Comparable) {
   . . .
```

Interfaces Properties:

- Interfaces can have hierarchies just like classes.
  - For example the interface **`Collection`** includes a **`contains`** method:

```
public interface Collection {
  . . .
   boolean contains(Object o);
```

  - The **`List`** interface extends the **`Collection`** interface and includes a **`lastIndexOf`** method.

```
public interface List extends Collection {
  . . .
   int lastIndexOf(Object o);
```

- Any class the implements the **`List`** interface, must have definitions for all methods in the **`Collection`** interface as well.

<u>Interfaces Properties</u>:

- A class can have just one parent class, but it can implement multiple interfaces
  - For example, the if the **`Employee`** class implements the Comparable, and Cloneable interfaces, it's header would be:

```
public class SalesPerson implements Clonable, Comparable {
```

  - Interfaces seem to be the answer to C++ 's multiple inheritance.