

## Abstract Classes and Interfaces

Developing new classes is easier after you understand the concept of inheritance. When you extend a class, the subclass inherits all the general attributes already defined in the base class; thus, you must create only the new, more specific attributes for the subclass.

For example, a *SalariedEmployee* and an *HourlyEmployee* are more specific than an *Employee*. They can inherit general Employee attributes, such as an employee number, but they add specific attributes, such as unique pay-calculating methods.

A superclass contains the features that are shared by all of its subclasses. For example, the attributes of the Dog class are shared by every Poodle and Spaniel. The subclasses are more specific examples of the superclass type; they add more features to the shared, general features. Conversely, when you examine a subclass, you see that its parent is more general and less specific; for example, Animal is more general than Dog. As you move up the inheritance hierarchy the more general the objects become.

We would want to avoid the need to redefine basic attributes like name, address, date of hire and other general data. We would also want to be able to generate a new class quickly that would include all of those fields. This is the nature of inheritance; it allows quick reuse and the organization of objects into a general classification.

A **concrete class** is one from which you can instantiate objects. Sometimes, a class is so general that you never intend to create any specific instances of the class, you just want to define a set of attributes and behaviors that can be shared by subclasses. For example, you might intend never to create an object that is “just” an Employee because the business you are writing the software for does not have a classification of *Employee*; the Employees are more specifically categorized around their type of payment {SalariedEmployee, HourlyEmployee, or ContractEmployee}. You know that you want to be able to define the general characteristics and behaviors in a common class so that they can be easily shared via the inheritance mechanism.

**You also want to ensure that actual instances of the very generic Employee class can never be created.**

A class such as Employee that you create only to extend from, and not directly instantiate is not a concrete class; it is an **abstract class**. Classes that you declare to be abstract are only able to be extended, not used directly. If you attempt to instantiate an object from an abstract class, you receive an error message from the compiler that you have committed an **InstantiationError**. An abstract class is the ultimate generalization for an inheritance hierarchy.

You use the keyword **abstract** when you declare an abstract class.

```

1  //·abstract·class·Cannot·be·instantiated
2  public·abstract·class·Employee·extends·Person·{
3      ....//·instance·variables
4      ....private·Date·....hireDate;
5      ....private·int·....id;
6      ....private·String·department;
7

```

The class above extends the Person class, which contains attributes like **name**, **date of birth** and **address**. The Employee class adds new fields to the hierarchy that pertain to being an employee in the company. It is also marked as **abstract** which means it **can only be extended**.

**Abstract classes can include two method types:**

- **Non-abstract (concrete) methods**, like those you can create in any class, are implemented in the abstract class and are simply inherited by its children.
- **Abstract methods** have no code or definition body and **must be implemented** in child classes.

Abstract classes usually contain at least one abstract method. When you create an abstract method, you provide the keyword **abstract** and the rest of the method header, including the method type, name, and parameters. However, the declaration ends there: you do not provide curly braces or any statements within the method—just a semicolon at the end of the declaration. This creates a situation where a **subclass is guaranteed to have the method and must override the method**.

Our Employee class has a single abstract method: **getPay()**

```

•//·abstract·method,·with·no·implementation·Class·muct·be·abstract
•//·there's·no·way·to·know·a·generic·Employee's·pay
•//·but·ALL·employees·need·to·be·able·to·get·their·pay
•//·implementation·will·be·provided·in·sub·class
•public·abstract·double·getPay();

```

**This method is marked as abstract for three reasons**

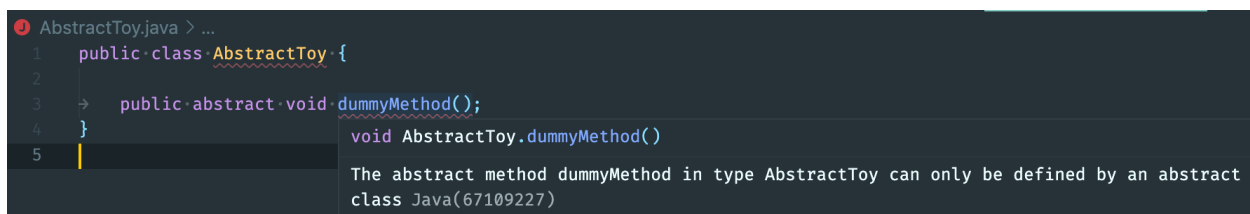
1. We don't know how to calculate a regular Employee's pay because those details are not included at the Employee level of the inheritance hierarchy.

2. We want to make certain that any class that extends Employee **must implement this method** and provide the details for how it will function. This is the real reason behind including it at this level: to ensure that all subclasses will be guaranteed to have this method. The compiler will enforce the implementation of **getPay()** for any subclass of Employee.
3. We want to be able to use polymorphism to easily calculate the pay for any subclass of Employee without the need for **tedious type checks**. This allows for infinite future extension without modifying the code of the abstract method!

### When making abstract declarations:

- If you declare a **class** to be abstract, there can be a mixture of abstract and concrete methods.
- If you declare a single method to be abstract, you **must** also declare its class to be abstract.
- Abstract classes can have regular constructor methods, but they cannot be called in conjunction with **new**. They can only be implicitly referenced via a **super** call in a subclass' constructor.

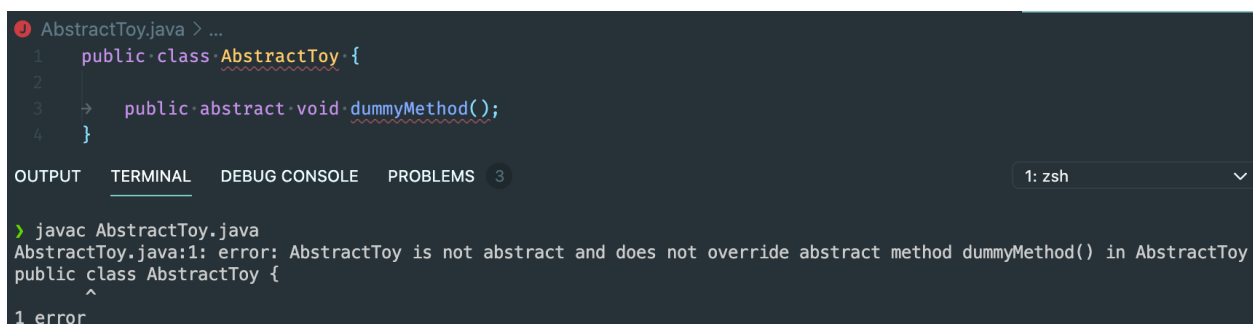
Let's see some examples of the Java compiler complaining about these concepts. Here is a toy example



```
AbstractToy.java > ...
1 public class AbstractToy {
2
3   public abstract void dummyMethod();
4 }
5
```

The abstract method dummyMethod in type AbstractToy can only be defined by an abstract class Java(67109227)

Even before attempting to compile this file VS Code issues warnings. What will the compiler say?



```
AbstractToy.java > ...
1 public class AbstractToy {
2
3   public abstract void dummyMethod();
4 }
```

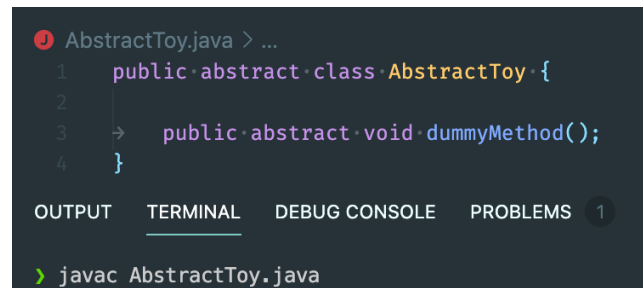
OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 3 1: zsh

```
> javac AbstractToy.java
AbstractToy.java:1: error: AbstractToy is not abstract and does not override abstract method dummyMethod() in AbstractToy
public class AbstractToy {
^
1 error
```

The compiler issues an error message saying that the class **AbstractToy** is not abstract and does not override the abstract method.

We have two options here:

1. Implement ***dummyMethod()*** in this class by providing a set of curly braces and some code in the method body
- or
2. Mark the class ***AbstractToy*** as ***abstract***



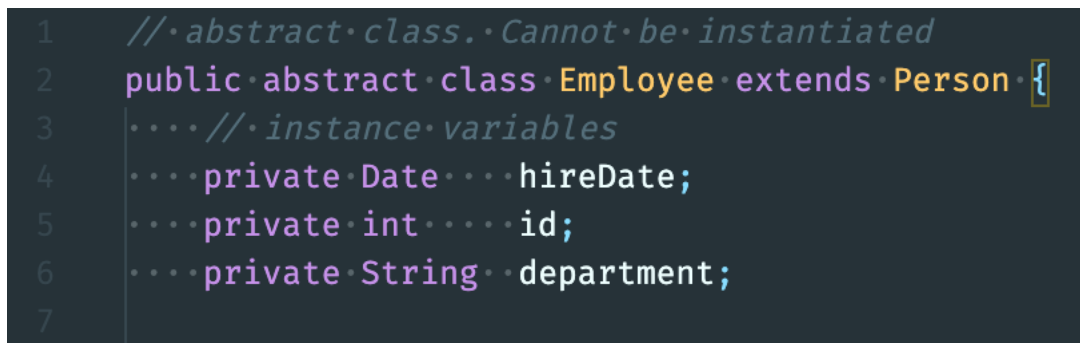
```
AbstractToy.java > ...
1 public abstract class AbstractToy {
2
3     → public abstract void dummyMethod();
4 }

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 1
> javac AbstractToy.java
```

## Cannot Instantiate Abstract Classes

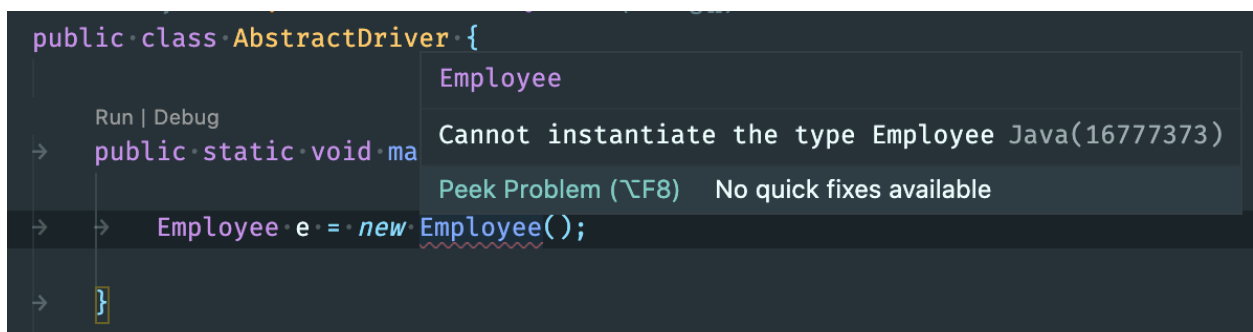
The Java compiler will not allow you to create an instance of an abstract class. This should make sense . . . why create an instance of a class that has methods with no implementation? What would those methods do when called?

I'll try to create an instance of the Employee abstract class shown above.



```
1 // abstract class Cannot be instantiated
2 public abstract class Employee extends Person {
3     ... // instance variables
4     ... private Date ... hireDate;
5     ... private int ... id;
6     ... private String ... department;
7 }
```

VS Code notices this straight away



```
public class AbstractDriver {
    Run | Debug
    → public static void main() {
        → Employee e = new Employee();
        → }
}
```

Employee  
Cannot instantiate the type Employee Java(16777373)  
Peek Problem (⌘F8) No quick fixes available

The compiler is definitely not happy about this situation

```
AbstractDriver.java > AbstractDriver > main(String[])
1 public class AbstractDriver {
2
3     Run | Debug
4     → public static void main(String[] args) {
5     →     Employee e = new Employee();
6
7     → }
8 }
```

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS 1

```
> javac AbstractDriver.java
AbstractDriver.java:5: error: Employee is abstract; cannot be instantiated
    Employee e = new Employee();
                   ^
1 error
```

## Overriding Abstract Methods

When you create a subclass that inherits an abstract method(s) you are **required to override every inherited abstract method**. Either the child class method must itself be abstract, or you must provide a body, or implementation, for the inherited method. Once this condition has been met you are free to instantiate subclass objects as normal.

The following example illustrates 2 Employee subclasses

1. **HourlyEmployee**: Pay is calculated based on a **wage** field and an **hoursWorked** field
2. **SalariedEmployee**: Pay is calculated based on a **salary** field

### HourlyEmployee

```
// HourlyEmployee extends an abstract class
// it MUST implement all abstract methods in super class
// or be declared abstract itself
public class HourlyEmployee extends Employee {
    ... private double wageRate;
    ... private double hours; // for the month
}
```

**HourlyEmployee** implementation of abstract superclass method **getPay()**. Notice the **@Override** annotation. Using this annotation tells the compiler to **enforce a proper override**

```
....//override abstract methods inherited from
....//abstract super class
....@Override
....public double getPay(){
....    return wageRate * hours;
....}
```

### SalariedEmployee

```
//SalariedEmployee extends an abstract class
//it MUST implement all abstract methods in super class
//or be declared abstract itself
public class SalariedEmployee extends Employee{
....private double salary; //annual
```

**SalariedEmployee** implementation of abstract superclass method **getPay()**. Notice the **@Override** annotation. Using this annotation tells the compiler to **enforce a proper override**

```
..//override abstract methods inherited from
..//abstract super class
..@Override
..public double getPay(){
..    return salary/12;
..}
```

After the successful override of the abstract method, **SalariedEmployee** and **HourlyEmployee** are considered **concrete** classes and objects of these classes can be instantiated as normal.

I will use the following **SalariedEmployee** constructor.

```
public SalariedEmployee(String firstName, String lastName, Date theDate, double theSalary){
....super(firstName, lastName, theDate, null, 0);
....if (theSalary >= 0)
....    salary = theSalary;
..}
```

And the following *HourlyEmployee* constructor

```
..public HourlyEmployee(.String firstName, String lastName,  
.....Date theDate, double theWageRate, double theHours){  
.....super(firstName, lastName, theDate, null, 0);  
.....this.setRate(theWageRate);  
.....this.setHours(theHours);  
..}
```

Notice that the constructors of the *Employee* class **can** be called by referencing *super*. Understand that this is a subtle difference from creating an explicit instance using *new*. You still need a way to get the appropriate data into the superclass even if it is abstract.

Below you will see two examples of this. I purposefully left a handful of fields null just for simplicity, and it is possible that employee management software may not constantly need data like **date of birth** and **address**. These are design choices that will be made by your team.

```
AbstractDriver.java > ...  
1 public class AbstractDriver {  
2  
3     Run | Debug  
4     public static void main(String[] args) {  
5         // SalariedEmployee instance  
6         SalariedEmployee se = new SalariedEmployee("Frank", "Stein", new Date(1, 21, 2021), 12345.67);  
7         System.out.println("\n" + se);  
8  
9         // HourlyEmployee instance  
10        HourlyEmployee he = new HourlyEmployee("Mary", "Shelley", new Date(2, 22, 2020), 15.75, 40);  
11        System.out.println("\n" + he);  
12    }  
13 }
```

OUTPUT    TERMINAL    DEBUG CONSOLE    PROBLEMS    1: zsh

```
> javac AbstractDriver.java  
> java AbstractDriver  
  
SALARIED EMPLOYEE: Frank Stein  
Born: null  
Address: null  
Hired: 1/21/2021  
ID: 0  
Dept: null  
$12345.67 per year  
  
HOURLY EMPLOYEE: Mary Shelley  
Born: null  
Address: null  
Hired: 2/22/2020  
ID: 0  
Dept: null  
$15.75 per hour for 40.0 hours
```

## Polymorphism and Abstract Classes

Now we are getting to the real advantages of these strange concepts. The abstract Employee class allows us to force subclasses to implement methods and provide class specific implementation. It also allows us to group all subclasses into a collection of the super class type so that we can process the entire Employee population around some shared behavior. In this case the shared behavior is the **getPay()** method. As we have seen, all sub classes of Employee must provide an implementation of this method, or the compiler will not process the code.

Let's take advantage of this by writing a method that will compute the total payroll amount for a collection of Employees. **We want to avoid having to perform unnecessary type checks.**

I have created a new subclass of Employee called **ContractEmployee**. The **getPay()** implementation for this class will be based on a single double field called **contractBid**. I used VS Code's **source action** tool to quickly generate all the constructors, getters/setters, toString and equals method. All I really had to do was ensure that there were no privacy leaks and then write the implementation for **getPay()**. The beauty of inheritance.

**Goal: Calculate total payroll across all Employee classifications**

```
public static double calculatePayroll(Employee[] employees){  
→ double payroll = 0;  
→ for (Employee employee : employees)  
→ → payroll += employee.getPay();  
→ return payroll;  
}
```

Polymorphic method call to **getPay()**

**This code deserves some explanation**

- The parameter to **calculatePayroll** is an array of **Employee objects**. Abstract classes (Employee in this case) can be used as the data type for a reference variable. The “**is a**” relationship between the superclass and subclasses dictates this.
- Remember that an **implicit upstream cast** occurs when an instance of a subclass type is assigned to variable of a superclass type. This can be a direct assignment using the assignment operator or an implicit assignment via a method call.
- The only methods that can be called through the Employee variable are methods explicitly defined in the Employee class or inherited from Employee's superclass. After an upstream cast the origin of the original object is “lost”, hence the need for downstream casts.
- Based on the concept of **dynamic method binding** and **polymorphism** the call to **getPay()** will resolve to the correct runtime type of the object. Remember that the



runtime type of an object is determined by the specific constructor that was used to initialize the object.

Runtime types

```
// SalariedEmployee instance
SalariedEmployee se = new SalariedEmployee("Frank", "Stein", new Date(1, 21, 2021), 12345.67);
// HourlyEmployee instance
HourlyEmployee he = new HourlyEmployee("Mary", "Shelley", new Date(2, 22, 2020), 15.75, 40);
// ContractEmployee instance
ContractEmployee ce = new ContractEmployee("Count", "Dracule", new Date(6, 3, 2018),
    "HR", 12345, 2300);
// Array of Employee objects. Triggers upstream type cast
Employee[] payrollGroup = {se, he, ce};
```

Upstream cast happens here

Only methods defined in class Employee can be called through reference variables of type Employee. Good thing that abstract method signature **getPay** is put up into class Employee!

Here is the output when the program is executed

```
1 public class AbstractDriver {
2
3     Run | Debug
4     public static void main(String[] args) {
5         // SalariedEmployee instance
6         SalariedEmployee se = new SalariedEmployee("Frank", "Stein", new Date(1, 21, 2021), 12345.67);
7         // HourlyEmployee instance
8         HourlyEmployee he = new HourlyEmployee("Mary", "Shelley", new Date(2, 22, 2020), 15.75, 40);
9         // ContractEmployee instance
10        ContractEmployee ce = new ContractEmployee("Count", "Dracule", new Date(6, 3, 2018),
11            "HR", 12345, 2300);
12        // Array of Employee objects. Triggers upstream type cast
13        Employee[] payrollGroup = {se, he, ce};
14        // call method and pass array of Employee objects
15        double total = calculatePayroll(payrollGroup);
16        System.out.printf("\nTotal payroll for all Employees is: $%.2f\n", total);
17    }
18
19    public static double calculatePayroll(Employee[] employees) {
20        double payroll = 0;
21        for (Employee employee : employees)
22            payroll += employee.getPay();
23        return payroll;
24    }
25 }
```

OUTPUT    TERMINAL    DEBUG CONSOLE    PROBLEMS    2: Java Debug Console

```
> javac AbstractDriver.java
> java AbstractDriver

Total payroll for all Employees is: $3958.81
```

The math checks out:

```
[jshell> (12345.67 / 12) + (15.75 * 40) + 2300;  
$2 ==> 3958.8058333333333
```

Let's add calls to ***getClass()*** and ***instanceof*** to ***calculatePayroll()*** to illustrate the association of these objects with their various types.

```
public static double calculatePayroll(Employee[] employees){  
→ double payroll = 0;  
→ for (Employee employee : employees){  
→ → // this is just for illustrating the various types of variable 'employee'  
→ → boolean isObject = employee instanceof Object;  
→ → boolean isPerson = employee instanceof Person;  
→ → boolean isEmployee = employee instanceof Employee;  
→ → boolean inHierarchy = isObject && isPerson && isEmployee;  
→ → // print a summary  
→ → System.out.println("\nIs 'employee' in the inheritance hierarchy? -> " + inHierarchy);  
→ → System.out.println("What is the runtime type of 'employee'? -> " + employee.getClass());  
→ → System.out.println("Which getPay() method will be called? -> " + employee.getClass());  
→ → payroll += employee.getPay();  
→ }  
→ return payroll;  
→ }
```

This addition produces the following output. You can see that ***getClass()*** returns the original runtime type of the ***employee*** variable, even after the ***upstream cast*** amnesia. This is the essence of ***dynamic method binding***.

```
> javac AbstractDriver.java  
> java AbstractDriver  
  
Is 'employee' in the inheritance hierarchy? -> true  
What is the runtime type of 'employee'? -> class SalariedEmployee  
Which getPay() method will be called? -> class SalariedEmployee  
  
Is 'employee' in the inheritance hierarchy? -> true  
What is the runtime type of 'employee'? -> class HourlyEmployee  
Which getPay() method will be called? -> class HourlyEmployee  
  
Is 'employee' in the inheritance hierarchy? -> true  
What is the runtime type of 'employee'? -> class ContractEmployee  
Which getPay() method will be called? -> class ContractEmployee  
  
Total payroll for all Employees is: $3958.81
```

You can also observe this behavior while running the code in debug mode.

During the first iteration of the for each loop you can see that the runtime type of the *employee* variable is ***SalariedEmployee***

```

VARIABLES
  Local
    > employees: Employee[3]@7
    payroll: 0.000000
    > employee: SalariedEmployee@9 ...
      addy: null
      birthDate: null
      department: null
      > firstName: "Frank"
      > hireDate: Date@22 "1/21/2021"
      id: 0
      > lastName: "Stein"
      salary: 12345.670000

  CALL STACK
    Thread [main] PAUSED ON STEP
      AbstractDriver.calculatePayroll
      AbstractDriver.main(String[])
    Thread [Reference ...] RUNNING
    Thread [Finalizer] RUNNING
    Thread [Signal Disp...] RUNNING

AbstractDriver.java > AbstractDriver > calculatePayroll(Employee[])
12  →  //·Array·of·Employee·objects··Triggers·upstream·type·cast
13  →  Employee[]·payrollGroup·=·{se,·he,·ce};
14  →  //·call·method·and·pass·array·of·Employee·objects
15  →  double·total·=·calculatePayroll(payrollGroup);
16  →  System.out.printf("\nTotal·payroll·for·all·Employees·is:·$%6.2f\n",·total);
17  →  }
18  →
19  →  public·static·double·calculatePayroll(Employee[]·employees){
20  →  double·payroll·=·0;
21  →  for·(Employee·employee·:·employees){
22  →  //·this·is·just·for·illustrating·the·various·types·of·variable·'employee'
23  →  boolean·isObject·→·=·employee·instanceof·Object;
24  →  boolean·isPerson·→·=·employee·instanceof·Person;
25  →  boolean·isEmployee·→·=·employee·instanceof·Employee;
26  →  boolean·inHierarchy·=·isObject·&&·isPerson·&&·isEmployee;
27  →  //·print·a·summary
28  →  System.out.println("\nIs·'employee'·in·the·inheritance·hierarchy?·→·"+·inHierarchy);
29  →  System.out.println("What·is·the·runtime·type·of·'employee'?·→·"+·employee.getClass());
30  →  System.out.println("Which·getPay()·method·will·be·called?·→·"+·employee.getClass());
31  →  payroll·+=·employee.getPay();
32  →  }
33  →  return·payroll;
34  →  }
35  →  }

```

The second iteration shows that the runtime type changes to ***HourlyEmployee***

```

VARIABLES
  Local
    > employees: Employee[3]@7
    payroll: 1028.805833
    > employee: HourlyEmployee@46 ...
      addy: null
      birthDate: null
      department: null
      > firstName: "Mary"
      > hireDate: Date@50 "2/22/2020"
      hours: 40.000000
      id: 0
      > lastName: "Shelley"
      wageRate: 15.750000

  CALL STACK
    Thread [main] PAUSED ON STEP
      AbstractDriver.calculatePayroll
      AbstractDriver.main(String[])
    Thread [Reference ...] RUNNING
    Thread [Finalizer] RUNNING
    Thread [Signal Disp...] RUNNING

AbstractDriver.java > AbstractDriver > calculatePayroll(Employee[])
12  →  //·Array·of·Employee·objects··Triggers·upstream·type·cast
13  →  Employee[]·payrollGroup·=·{se,·he,·ce};
14  →  //·call·method·and·pass·array·of·Employee·objects
15  →  double·total·=·calculatePayroll(payrollGroup);
16  →  System.out.printf("\nTotal·payroll·for·all·Employees·is:·$%6.2f\n",·total);
17  →  }
18  →
19  →  public·static·double·calculatePayroll(Employee[]·employees){
20  →  double·payroll·=·0;
21  →  for·(Employee·employee·:·employees){
22  →  //·this·is·just·for·illustrating·the·various·types·of·variable·'employee'
23  →  boolean·isObject·→·=·employee·instanceof·Object;
24  →  boolean·isPerson·→·=·employee·instanceof·Person;
25  →  boolean·isEmployee·→·=·employee·instanceof·Employee;
26  →  boolean·inHierarchy·=·isObject·&&·isPerson·&&·isEmployee;
27  →  //·print·a·summary
28  →  System.out.println("\nIs·'employee'·in·the·inheritance·hierarchy?·→·"+·inHierarchy);
29  →  System.out.println("What·is·the·runtime·type·of·'employee'?·→·"+·employee.getClass());
30  →  System.out.println("Which·getPay()·method·will·be·called?·→·"+·employee.getClass());
31  →  payroll·+=·employee.getPay();
32  →  }
33  →  return·payroll;
34  →  }
35  →  }

```

And the third iteration shows that the runtime type changes to **ContractEmployee**

The screenshot shows the VS Code interface with the Variables and Call Stack panels. The Variables panel on the left shows the state of the program. The 'employee' variable is highlighted, showing its runtime type as 'ContractEmployee@67...' and its fields: 'addy: null', 'birthDate: null', 'contractBid: 2300.000000', 'department: "HR"', 'firstName: "Count"', 'hireDate: Date@72 "6/3/2018"', 'id: 12345', and 'lastName: "Dracule"'. The Call Stack panel on the right shows the execution flow, with the current step being 'AbstractDriver.calculatePayroll'.

This order of runtime types logically corresponds to the in order in which the Employee objects were added to the array . . . shown here.

```
// SalariedEmployee instance
SalariedEmployee se = new SalariedEmployee("Frank", "Stein", new Date(1, 21, 2021), 12345.67);
// HourlyEmployee instance
HourlyEmployee he = new HourlyEmployee("Mary", "Shelley", new Date(2, 22, 2020), 15.75, 40);
// ContractEmployee instance
ContractEmployee ce = new ContractEmployee("Count", "Dracule", new Date(6, 3, 2018),
    "HR", 12345, 2300);
// Array of Employee objects. Triggers upstream type cast
Employee[] payrollGroup = {se, he, ce};
```

## Cannot call unique (non-inherited) subclass methods from superclass reference

As this heading states, when you have a reference to a superclass object you can only call methods that are defined in that class, or *it's superclass*. This is why we decided to place the **getPay()** signature at the Employee level . . . so we can generically call it across all unique subclass types.

For example, you cannot call the **ContractEmployee** method **getContractBid()** from this generic context. VS Code immediately identifies this as a problem.

```
public static double calculatePayroll(Employee[] employees){
    double payroll = 0;
    for (Employee employee : employees) {
        // try to call method
        double bid = employee.getContractBid();
        payroll += employee.getPay();
    }
    return payroll;
}
```

The method getContractBid() is undefined for the type Employee

Peek Problem (⌘F8) Quick Fix... (⌘.)

And the compiler also has a problem with this. The method does in fact exist, just not at the Employee level. We place methods and fields high in the inheritance hierarchy so they can be shared and polymorphically called across all sub types. This is a key part of object-oriented design.

```
19 → public static double calculatePayroll(Employee[] employees){
20 → → double payroll = 0;
21 → → for (Employee employee : employees){
22 → → → // try to call method NOT IN EMPLOYEE class
23 → → → double bid = employee.getContractBid();
24 → → → payroll += employee.getPay();
25 → → → }
26 → → return payroll;
27 → }
28 }
29
```

OUTPUT   TERMINAL   DEBUG CONSOLE   PROBLEMS 1

```
> javac AbstractDriver.java
AbstractDriver.java:23: error: cannot find symbol
    double bid = employee.getContractBid();
                           ^
symbol:   method getContractBid()
location: variable employee of type Employee
1 error
```