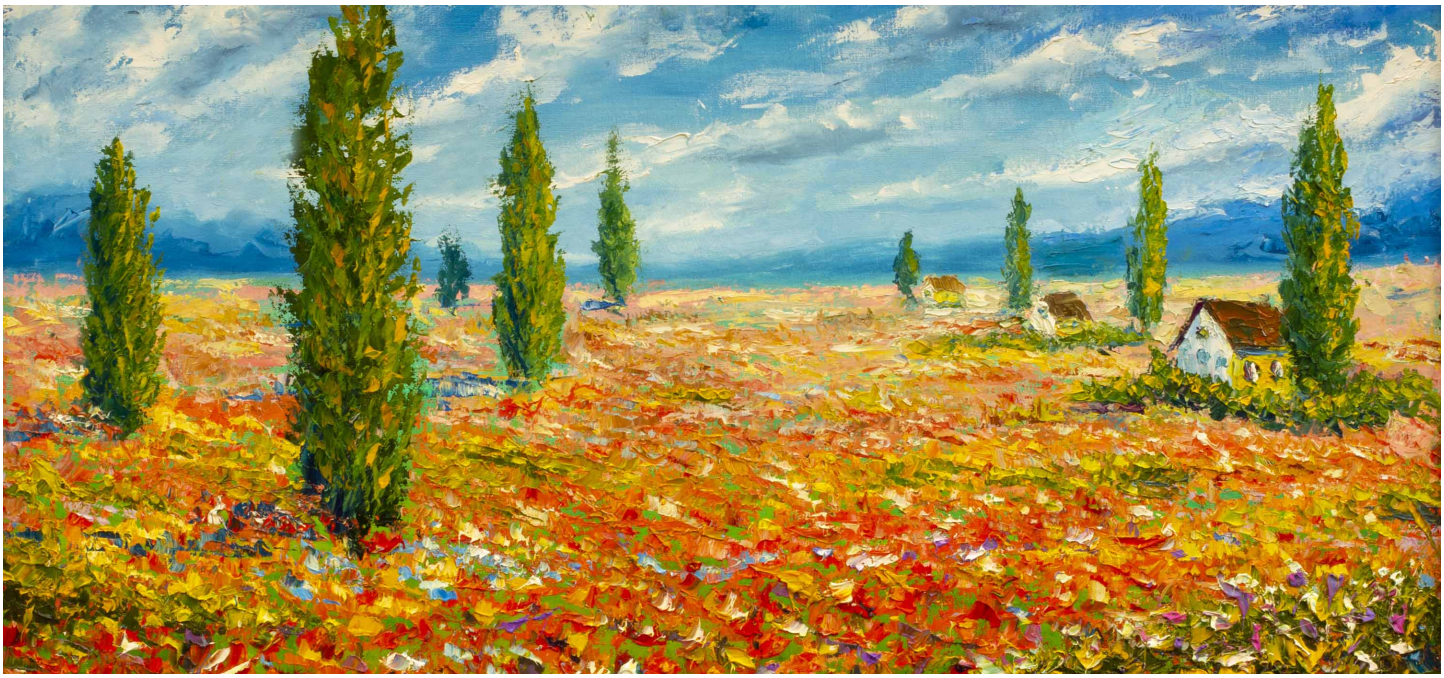


10-Instance Variables & Constructors

Author: Vincent Lau

Learning Objectives

> Create classes with constructors, and getters and setters
>
> Describe the use of access modifiers
>
> Differentiate a class and an object
>
> Create and initialize an object
>
> Understand the two main reasons to use this keyword

Classes

An Example of Class

- A class is a blueprint or prototype from which objects are created.

```java
public class Person {
    // The Person class has four fields
    private String name;
    private int age;
    private int height;
    private int weight;

    // The Person class now has one constructor
    public Person(String name, int age, int height, int weight) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
    }

    // The Person class has four methods
    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    public void printState() {
        System.out.println("name: " + this.name +
                " age: " + this.age +
                " height: " + this.height +
                " weight: " + this.weight);
    }
}
```

- A class declaration for *Firefighter* class that is a subclass of *Person* might look like this:

```java
public class Firefighter extends Person {
    // the Firefighter subclass has one field
```

```
    private int hoursOfTraining;

    // the Firefighter subclass has one constructor
    public Firefighter(String name, int age, int height,
                       int weight, int hoursOfTraining) {
        // super method
        // Constructor of parent class: public Person(String name, int age, int height, int weight)
        super(name, age, height, weight);
        this.hoursOfTraining = hoursOfTraining;
    }

    // the Firefighter subclass has one method
    public void setHoursOfTraining(int hoursOfTraining) {
        this.hoursOfTraining = hoursOfTraining;
    }

    public static void main(String[] args) {
        Firefigther f1 = new Firefigther("John", 23, 170, 70, 200);
        f1.getHeight();
    }
}
```

- *Firefighter* inherits all the fields and methods of *Person* and adds the field *hoursOfTraining* and a `setter method`.

Declaring Classes

- This is a *class declaration.*

```
class SomeClass {
    // field
    // constructor
    // method declarations
}
```

- The *class body* (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class:
  - **`Constructors`** - initialize new objects
  - **`Declarations`** the fields - provide the state of the class and its objects
  - **`Methods`** - implement the behavior of the class and its objects

- A subclass that inherits a class and implements an interface looks like:

```
class Cat extends Animal implements SomeInterface, SomeInterface2 {
                // field, constructor, and method declarations
}
```

- You can also add modifiers like *public* or *private* before the *class* keyword. Publicly accessible classes use the *public* modifier, while `Nested Classes` use *private*.
- In general, class declarations can include these components, in order:

Component	Description
Modifiers	Added before the keyword class, such as public and private, to control access.
Class name	The initial letter is capitalized by convention.
Name of the parent class	Always preceded by the keyword **extends**. A class can only extend one parent.
A **comma**-separated list of interfaces implemented by the class	Always preceded by the keyword implements. A class can implement **more than one interface**.
The class body	Always surrounded by braces { }.

Declaring Instance Variables

- The *Person* class defines its fields like this:

```
private String name;
private int age;
private int height;
private int weight;
```

- Fields declarations are composed of three components, in order:
  - **[Technically Optional]** Access modifiers, such as *public* or *private*
  - **[Mandatory]** The field's type
  - **[Mandatory]** The field's name

```
```

```
1 private int salary;
```

Default Values

1. `int` : The default value for `int` is `0` .
2. `double` : The default value for `double` is `0.0` .
3. `char` : The default value for `char` is the null character `'\u0000'` (or simply `'\0'`).

```
1 public static void main(String[] args) {
2     char test = '\u0000';
3     System.out.println("start" + test + "end"); // print startend
4 }
```

4. `boolean` : The default value for `boolean` is `false` .
5. `float` : The default value for `float` is `0.0f` .
6. `byte` : The default value for `byte` is `0` .
7. `short` : The default value for `short` is `0` .
8. `long` : The default value for `long` is `0L` .

Access Modifiers

- The fields of *Person* all have the `private` modifier. That means they are only accessible within the class itself. Using `public` modifier would mean the field being accessible from all other classes.
- Using `private` is preferred over using *public,* especially for member variables, because the fields should not be *directly* exposed to any objects.

```
1 public class Person {
2     public String name;
3     private int age;
4     private int height;
5     private int weight;
6 }
```

Reserved Keywords

- Here are the Java keywords and reserved words that can not be used as names for Java variables.

abstract	default	goto	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	null	synchronized	

Variable Naming Convention

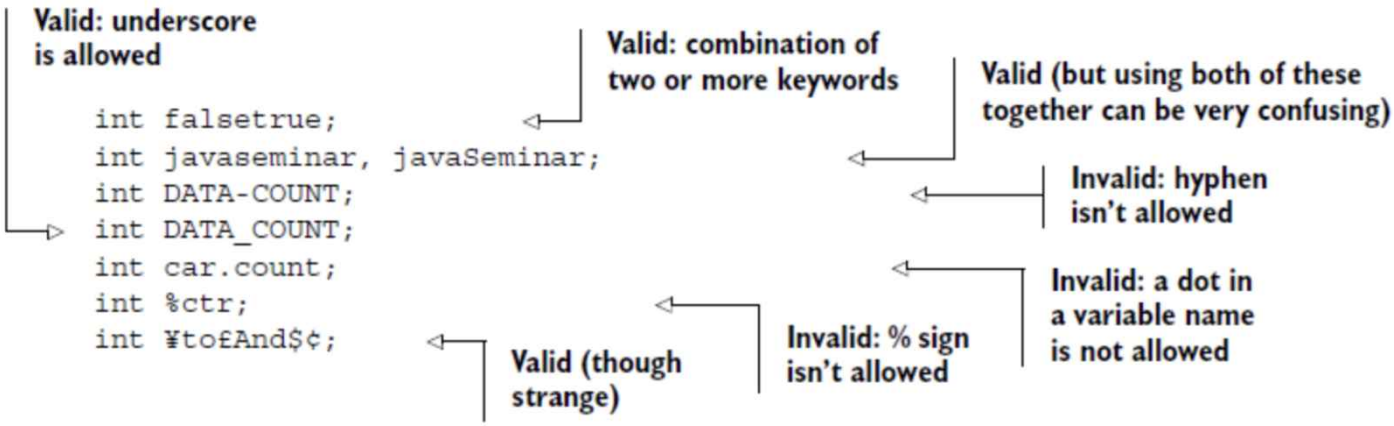
- The name of a variable should express the intent and be self-explanatory.

```
1 int itemsPurchased = 100; // camal-case
2 final int MAXIMUM_AMOUNT_OF_INVENTORY = 200; // uppercase snake case
3
4 int i = 0; // bad and unexpressive naming
```

Naming conventions in Java

NAMING CONVENTIONS	APPLICATION	EXAMPLES
Lower Camel Case	variables and methods	firstName timeToFirstLoad indexNumber
Upper Camel Case	classes, interfaces, annotations, enums, records	TomcatServer RestController WriteOperation
Screaming Snake Case	constants	INTEREST_RATE MINIMUM_SALARY EXTRA_SAUCE
lower dot case	packages and property files	java.net.http java.management.rmi application.properties
kebab case	not recommended	landing-page.html game-results.jsp 404-error-page.jsf

Common Mistakes



Defining Constructors

- **Constructors are used to create objects** from the class blueprint. Constructor declarations look like method declarations - except they use the name of the class and have no return type.

```
1 public Person(String name, int age, int height, int weight) {
2     this.name = name;
3     this.age = age;
4     this.height = height;
5     this.weight = weight;
```

```
6 }
```

- To create a *Person* object called *somebody,* the constructor is called by the *new* operator.

```
1 // "Person somebody" declares the object reference variable
2 // while "new Person("Somebody", 18, 180, 75)" creates space in memory for the
  object
3 // and initializes its fields
4 Person somebody = new Person("Mary", 18, 180, 75);
```

- Although *Person* only has one constructor, it could have others, including a no-argument constructor

```
1 public Person() {
2     this.name = "Unknown"; //default value
3     this.age = 0;
4     this.height = 0;
5     this.weight = 0;
6 }
```

- Then the **no-argument constructor** can be invoked to create a new *Person* object:

```
1 Person unknownPerson = new Person();
```

- You are allowed to declare **multiple constructors** as long as they don't clash.
- You cannot write two constructors with the same number and type of arguments for the same class, because the compiler will not be able to tell them apart. A compile-time error will be raised.
- **The compiler automatically provides a no-argument, default constructor for any class without constructors**. This default constructor will call the no-argument constructor of the superclass, be sure that the superclass has a no-argument constructor to invoke.
- If your class has no superclass, its no-argument default constructor will call the no-args constructor of the *Object* class. **In Java, all classes *implicitly extends* the *Object* class**, which has a no-argument constructor.

```
1 public class Person {
```

```
    private String name;
    private int age;
    private int height;
    private int weight;

   // Person() {}

    public Person(String name, int age, int height, int weight) {
        this.name = name;
        this.age = age;
        this.height = height;
        this.weight = weight;
    }
    // method here ...
}

/** COMPILATION ERROR **/
public class Firefighter extends Person {
    private int hoursOfTraining;

  //  public Person() {
  //      super();
  //  }

    // No explicit constructor has been defined in the Firefighter subclass
    // So, then the compiler automatically provide a no-args default constructor
    // But Person has an explicit constructor, then default no-arg contructor will NOT be generated

    // If we remove the explicit constructor, the Firefighter will be compiled correctly

    // the Firefighter subclass has one method
    public void setHoursOfTraining(int hoursOfTraining) {
        this.hoursOfTraining = hoursOfTraining;
    }
}
```

- You can also use access modifier in a constructor's declaration to control which other classes get to call the constructor in order to create objects of the class.

Using *this* keyword

- Within an instance method or a constructor, `this` is a reference to the current object - the object whose method or constructor is being called.
- You can refer to any member (including variables and methods) of the current object from within an instance method or a constructor by using *this*.
- There are mainly two scenarios for using *this* keyword:
  a. It is used so that the compiler can differentiate between the constructor parameters and the instance variables to be set.
  b. From within a constructor, you can also use the `this` keyword to call another constructor in the same class.
     - The invocation of another constructor must be the first line in the current constructor.

```java
// The constructor of the Person class
public Person(String name, int age, int height, int weight) {
    this.name = name;
    this.age = age;
    this.height = height;
    this.weight = weight;
}

// Assume it now has one more constructor
public Person(String name, int age, int height, int weight, LocalDate dateOfBirth) {
    this(name, age, height, weight); // first line of constructor to invoke another constructor
    this.dateOfBirth = dateOfBirth;
}
```

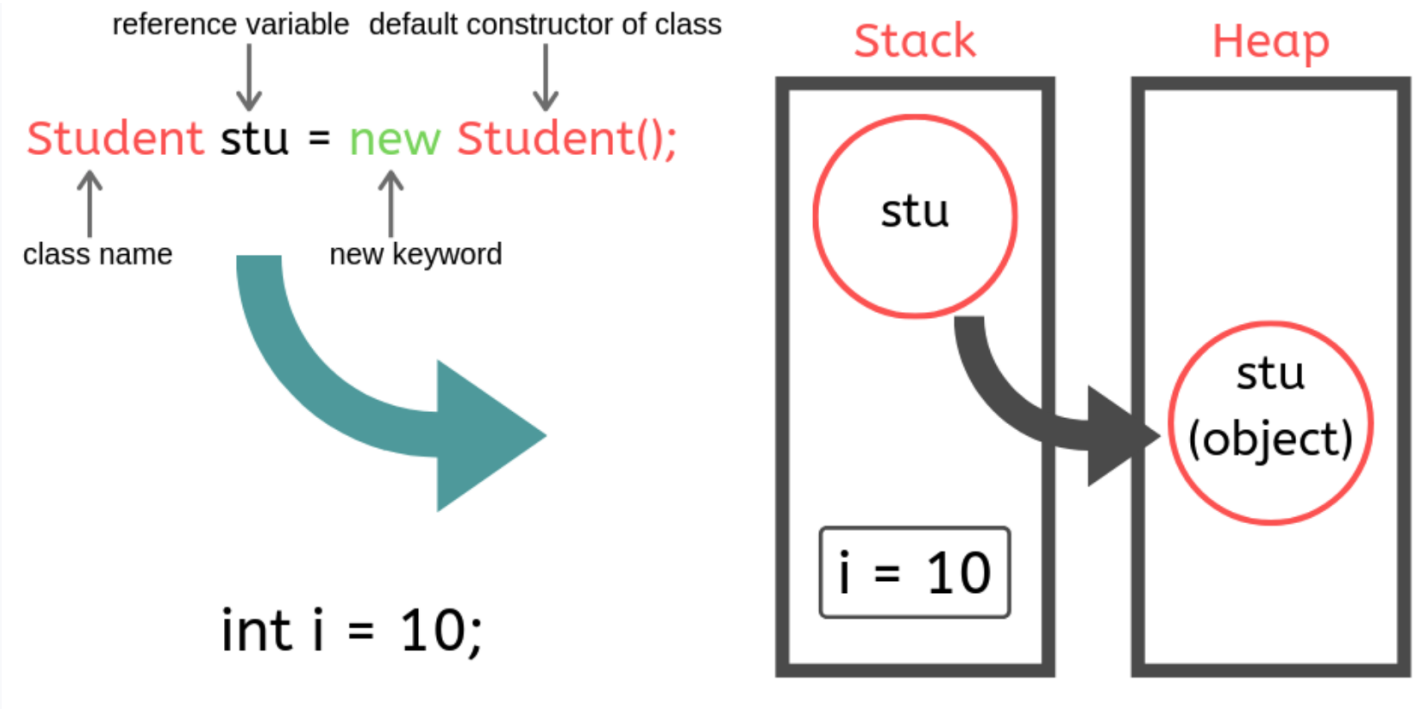
Objects

Creating Objects

- Creating an object involves three parts:
  a. **Declaration** - The object type is associated with a variable name
  b. **Instantiation** - The ***new*** operator is used to create the object in Java **(Java Keyword)**
  c. **Initialization** - The `constructor after the new operator` initializes the new object
- Simply declaring a reference variable does not create an object.

```java
public class Student {
    // instance variable with initial value 10
    int i = 10;

    public static void main(String[] args) {
        // You can spilit it into 2 lines, same thing.
        // Student stu; // Simply declaring a reference variable
        // stu = new Student(); // creates an object
        Student stu = new Student();
    }
}
```



- **stu** contains the reference to access the object.
- Stack memory contains the reference to the object.
- Inside RAM, when we create an object it’ s always stored in the heap space.
- Java automatic garbage collector runs on **heap** memory.
- When stack memory is full, Java throws **java.lang.StackOverflowError.**
- When heap memory is full, Java throws **java.lang.OutOfMemoryError.**

Calling an Object's Methods

```java
LocalDate dateOfBirth = LocalDate.of(1992, 8, 12);
Person person1 = new Person("Jane Chan", 25, 170, 55, dateOfBirth);

System.out.println(person1.getName());  // prints "Jane Doe"
```

```
5 System.out.println(person1.getDateOfBirth());  // prints "1997-07-01"
```

Access Modifiers

- Access modifiers determine whether other classes can access a given class and its fields, constructors, and methods.
- Access modifiers can be specified separately for a class, its constructors, fields, and methods.
- There are two levels of access control:
  - At the top level *public*, or package-private (if no explicit modifier)
  - At the member level *public, private, protected, or* package-private (if no explicit modifier)

Access Modifier	Within Class	Within Package	Same Package by subclasses	Outside Package by subclasses	Global
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

- At the top level (apply to *class* or *interface*):
  - **public:** in which case it is visible to all classes everywhere
  - **default(no modifier):** it is visible only within its own package.
- At the member level (apply to the body of *class* or *interface*):
  - **private:** A private modifier's access level is limited to the class. It isn't accessible outside of class.
  - **default(no modifier):** A default modifier's access level is limited to the package. It's not possible to get to it from outside the package. If you don't indicate an access level, the default will be used.
  - **protected:** A protected modifier's access level is both within the package and outside the package via a child class. The child class cannot be accessed from outside the package unless it is created.
  - **Public:** A public modifier's access level is visible everywhere. It can be accessible from within and outside the class, from within and outside the package, and from within and outside the package.

Pocket Money Example

```
1
2  // In the *GenerationX* package
3  package ProtectedDemo.GenerationX;
4
5  import java.math.BigDecimal;
6
7  public class FamilyA {
8      private static BigDecimal cashAmount = new BigDecimal("10000");
9      private static final BigDecimal POCKET_MONEY_RATIO = new
   BigDecimal("0.05");
10     private static final BigDecimal RED_ENVELOPE_RATIO = new
   BigDecimal("0.01");
11
12     // Accessible only to FamilyA's subclasses or same package
13     protected BigDecimal givePocketMoney() {
14         return cashAmount.multiply(POCKET_MONEY_RATIO);
15     }
16     // Accessible to all other classes
17     public BigDecimal giveRedEnvelopeMoney()  {
18         return cashAmount.multiply(RED_ENVELOPE_RATIO);
19     }
20 }
21
22 // A class that *extends* FamilyA from a different package
23 package ProtectedDemo.GenerationY;
24
25 import ProtectedDemo.GenerationX.FamilyA;
26 import java.math.BigDecimal;
27
28 public class FamilyAChild extends FamilyA {
29     public BigDecimal getPocketMoney() {
30         return super.givePocketMoney();  // equivalent to
   *super.givePocketMoney()*
31     }
32 }
33
34 // A class that does NOT *extend* FamilyA from a different package
35 package ProtectedDemo.GenerationY;
36
37 import ProtectedDemo.GenerationX.FamilyA;
38 import java.math.BigDecimal;
39
40 public class FamilyBChild { // not a subclass of FamilyA
```

```
41      // Cannot access FamilyA's *givePocketMoney* method, due to diff. package
    & not subclass
42      // Can only access *giveRedEnvelopeMoney* method
43      public BigDecimal getPocketMoney(FamilyA familyA) {
44          return familyA.givePocketMoney(); // compile error
45      }
46  }
```

Tips on Choosing an Access Level

- Use the most restrictive access level that makes sense for a particular member. Use `private` unless you have a good reason not to.
- Avoid *public* fields except for constants. Public fields tend to tightly couple you to a particular implementation and limit your flexibility in changing your code.

Questions

- Why the *private* modifier is preferred for instance variables?
- What is the difference between a class and an object?
- What does the compiler do if no constructor is provided for a class?
- What are the two use cases of the *this* keyword?