
Objects and Classes

- ☐ Functions
- ☐ Objects and Classes
- ☐ Inheritance
- ☐ Interfaces
- ☐ Packages
- ☐ Generics

Functions

Overview

- ▶ A function is a segment of code that performs a task and is extracted separately
 - ▶ Identified by a name
 - ▶ May be used (called) from different parts of the program
 - ▶ May have parameters (arguments, or inputs)
 - ▶ May return something (result, or output)
- ▶ Benefits:
 - ▶ Functions help to break down a large program into smaller problems
 - ▶ Functions reuse code
 - ▶ Functions enhance readability and maintainability
- ▶ In Java, every lines of code must be inside a class → any function needs to be member of some class, and is called a method

Function Example

```
class FunctionExample {  
    static double round2Decimals(double x) {  
        double y = x * 100.;  
        long z = Math.round(y);  
        return z / 100.;  
    }  
  
    public static void main(String argv[]) {  
        System.out.println(round2Decimals(Math.PI));    // 3.14  
        System.out.println(round2Decimals(Math.E));    // 2.72  
    }  
}
```

- ▶ **return** statement is used to terminate a function and return a value
- ▶ Existence of parameters and local variables is limited in the scope of the function that they are declared

Void Functions

- ▶ Are functions that don't return any result on termination

- ▶ `return` statement has no argument
- ▶ `return` statement at the end of function can be omitted

- ▶ Example:

```
class VoidFunctionExample {  
    static void printWelcome(String name) {  
        System.out.println("Hello " + name);  
    }  
  
    public static void main(String argv[]) {  
        printWelcome("John");  
        printWelcome("Bob");  
    }  
}
```

Object and Classes

Stateful Functions

- ▶ This example is about functions that change the state of the program, so that they will return a different result every time being called

```
▶ class SingleAccountApp {  
    static double balance = 0.;  
  
    static double deposit(double amount) {  
        balance += amount;  
        return balance;  
    }  
  
    static double withdraw(double amount) {  
        balance -= amount;  
        return balance;  
    }  
  
    public static void main(String argv[]) {  
        System.out.println(deposit(20.5)); // 20.5  
        System.out.println(withdraw(10)); // 10.5  
        System.out.println(deposit(15)); // 25.5  
    }  
}
```

Multiple Entities?

- ▶ Now, what if we want these functions to work with multiple bank accounts?
 - ▶ We might want to bind the balance and the functions into one container → this is where objects and classes come into play

```
// BankAccount.java
class BankAccount {           // define a class; remark that the members are now non-static
    double balance = 0.;

    double deposit(double amount) {
        return balance += amount;
    }

    double withdraw(double amount) {
        return balance -= amount;
    }
}

// MultipleAccountApp.java
class MultipleAccountApp {
    public static void main(String argv[]) {
        BankAccount account1 = new BankAccount(), // create 2 instances (objects) of BankAccount
        account2 = new BankAccount();

        System.out.println(account1.deposit(20.5)); // 20.5
        System.out.println(account1.withdraw(10)); // 10.5
        System.out.println(account2.deposit(15)); // 15 (not 25.5)
    }
}
```


Object-Oriented Programming (OOP)

- ▶ From real world ...
 - ▶ An object is thing, fact, entity,... which is characterized by its properties and can perform certain operations. Ex:
 - ▶ A student is an object characterized by: name, age, department, class, year,... and can perform: studying, doing exercises, going to class, doing tests,...
 - ▶ A cellphone is an object characterized by: SIM number, model, size,... and can perform: making call, texting, answering to calls, denying calls,...
 - ▶ A class is the description of properties and operations for some kind of objects
 - ▶ Simpler consideration: students are objects while the definition of students is a class, similarly for cellphones and their definition
- ▶ ... to programming:
 - ▶ A class is a data type, which basically encapsulates its properties and methods
 - ▶ Usually defined in a .java file and with the same name with the class
 - ▶ An object is an instance created with the type of the given class

Class Properties and Methods

- ▶ Properties = member variables, methods = member functions

- ▶ The methods can read value of the properties and update them

- ▶ Example

```
// Circle.java
class Circle {
    double radius;

    void setRadius(double newRadius) {
        radius = newRadius;
    }

    double getRadius() {
        return radius;
    }

    double getArea() {
        return Math.PI * radius * radius;
    }

    double getPerimeter() {
        return Math.PI * 2 * radius;
    }
}
```

- ▶ Using the class

```
// CircleExample.java
class CircleExample {
    public static void main(String args[]) {
        Circle c1 = new Circle();
        c1.setRadius(10);
        System.out.println(c1.getArea());

        Circle c2 = new Circle();
        c2.setRadius(20);
        System.out.println(c2.getPerimeter());
    }
}
```

Visibility of Members

- ▶ By default, variables and methods are accessible to other members of the class itself and to other classes in the same package
- ▶ Use a modifiers `public` and `private` to change the default behavior:

Modifier	Same class	Class in same package	Class in other packages
None (default)	✓	✓	✗
<code>public</code>	✓	✓	✓
<code>private</code>	✓	✗	✗

- ▶ From the OOP point of view, it's a good practice to declare class properties as private, and provide public setter and getter methods, for better encapsulation:
 - ▶ Possibility for read-only and write-only properties
 - ▶ Possibility for virtual properties
 - ▶ Programmer may make changes to the class in the future without affecting other parts of code

Visibility Example

```
public class Circle {  
    private double radius;  
  
    public void setRadius(double newRadius) {  
        radius = newRadius;  
    }  
  
    public double getRadius() {  
        return radius;  
    }  
  
    public void setDiameter(double newDiameter) { // virtual property  
        radius = newDiameter / 2.;  
    }  
  
    public double getDiameter() {  
        return radius * 2.;  
    }  
  
    public double getArea() {  
        return Math.PI * radius * radius;  
    }  
  
    public double getPerimeter() {  
        return Math.PI * 2 * radius;  
    }  
}
```

this

- ▶ Refers to the current object in a method or constructor
- ▶ The most common use is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter)

```
▶ public class Circle {  
    private double radius;  
  
    public void setRadius(double radius) {  
        this.radius = radius;  
    }  
    //...  
}
```

- ▶ Can also be used to:
 - ▶ Invoke current class constructor/method
 - ▶ Return the current class object
 - ▶ Pass an argument in the constructor/method call

static Members

- ▶ A **static** member belongs to the class (not to any particular object)
 - ▶ **static** variables are initialized only once at the start of the execution
 - ▶ **static** methods have access only to other **static** members, but without using the class's object (instance)
 - ▶ **this** is not defined in **static** methods
- ▶ Think about **String** class's **format()**, **split()** methods

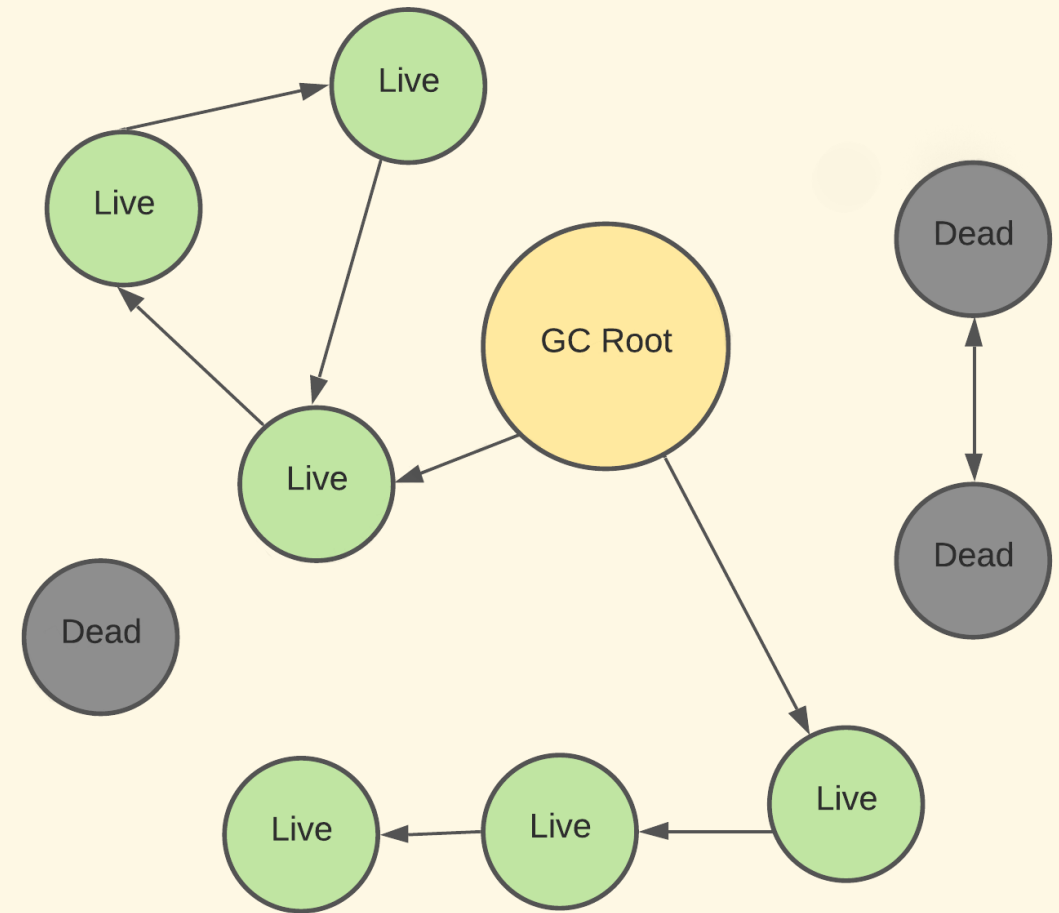
Constructors

- ▶ A constructor is a special method and is used to initialize new objects when being created
 - ▶ Is usually used to set initial values for objects properties
 - ▶ Has the same name of the class, does not have a return type, and can be overloaded
 - ▶ When no constructor is defined, the compiler automatically creates a default one (which is empty and takes no parameter)
- ▶ Example:

```
public class BankAccount {  
    private String name;  
    private double balance;  
  
    public BankAccount(String firstName) {  
        name = firstName;  
        balance = 0.;  
    }  
  
    public BankAccount(String firstName, double firstBalance) {  
        name = firstName;  
        balance = firstBalance;  
    }  
  
    // ...  
}
```

Garbage Collector (GC)

- ▶ GC is a background process for reclaiming the runtime unused memory by destroying the unused objects
 - ▶ Automatic detection of unused objects based on number of references to them (objects with circular references are also detected)
 - ▶ Programmers don't need to take care of destroying unused objects
 - ▶ GC is non-deterministic



`finalize()` Method

- ▶ Is a special method that is called by the GC before destroying the object from memory
- ▶ Is usually used to perform cleanup activity
- ▶ Is also non-deterministic
- ▶ Is deprecated in Java 9, because:
 - ▶ It's difficult to handle correct finalization for a group of related objects, since there is no ordering among `finalize()` methods; sometimes this may cause deadlocks
 - ▶ There are no guarantees about the timeliness of finalization
 - ▶ There is no explicit registration/deregistration mechanism
 - ▶ Alternatives: `java.lang.ref.Cleaner`, `java.lang.ref.PhantomReference`

null Value

- ▶ When an object variable does not reference any object, it holds a special literal value `null`
- ▶ Example:
 - ▶ `Student s1, s2 = new Student(); // s1 is null, s2 is not null`

`s1 = new Student(); // s1 now is not null`

`s2 = null; // s2 now is null,`
`// the created object is left unreferenced`

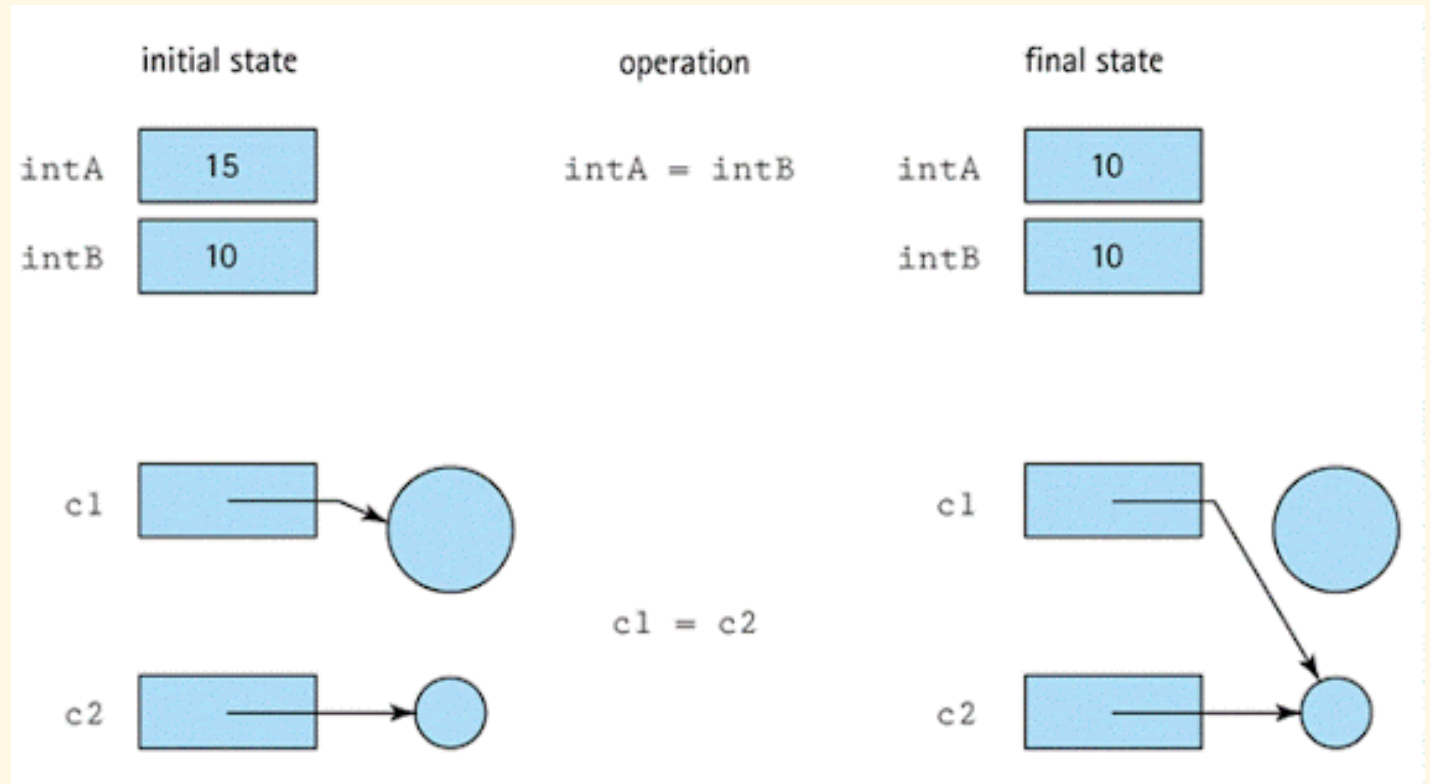
Difference in Usage of Primitive and Object Types

► Primitive types:

- Variables always have a value
- Assignment, comparison, parameter passing are done using value

► Object types (including strings, arrays):

- Variables may be `null`
- Assignment, comparison, parameter passing are done using reference



Wrapper Classes for Primitive Types

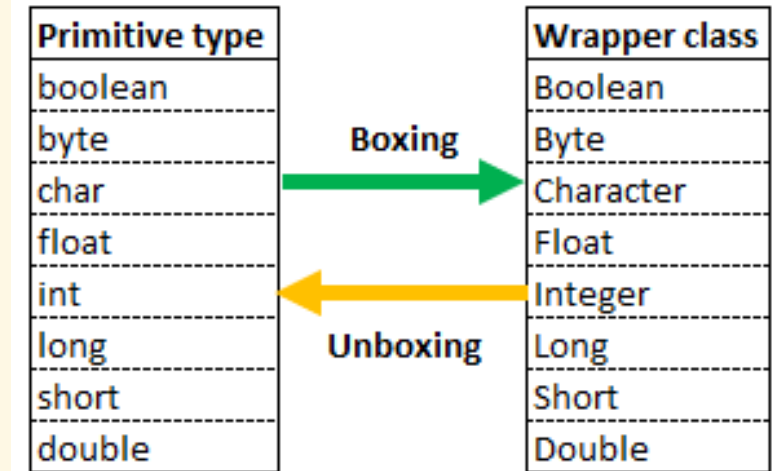
- ▶ Wrapper classes provide a way to use primitive types as objects, that is useful in
 - ▶ Passing parameters by reference
 - ▶ Working with **Collection** objects

- ▶ Example

```
Integer myInt = 5;           // boxing
Double myDouble = 5.99;
Character myChar = 'A';
System.out.println(myInt.intValue());
System.out.println(myDouble.doubleValue());
System.out.println(myChar.charValue());
```

```
int i = myInt;               // unboxing
```

- ▶ `ArrayList<int> myNumbers = new ArrayList<int>();` // Compile error
- ▶ `ArrayList<Integer> myNumbers = new ArrayList<Integer>();` // Ok!



Exercise

- Write a class allowing to work with complex numbers

```
MyComplex
- real:double = 0.0
- imag:double = 0.0
+ MyComplex()
+ MyComplex(real:double, imag:double)
+ getReal():double
+ setReal(real:double):void
+ getImag():double
+ setImag(imag:double):void
+ setValue(real:double, imag:double):void
+ toString():String •-----
+ isReal():boolean
+ isImaginary():boolean
+ equals(real:double, imag:double):boolean
+ equals(another:MyComplex):boolean
+ magnitude():double
+ argument():double •-----
+ add(right:MyComplex):MyComplex
+ addNew(right:MyComplex):MyComplex
+ subtract(right:MyComplex):MyComplex
+ subtractNew(right:MyComplex):MyComplex
+ multiply(right:MyComplex):MyComplex
+ divide(right:MyComplex):MyComplex
+ conjugate():MyComplex
```

"(real+imagi)", e.g., "(3+4i)"

In radians

add(), subtract(), multiply(), divide():
add/subtract/multiply/divide the given instance right into this instance, and return this instance.
addNew(), subtractNew(): add/subtract this and right, and return a new instance. this instance shall not be changed.
conjugate(): on this instance

Inheritance

Initial Example

- ▶ To manage the personnel of a company, we need to define classes corresponding to the posts:

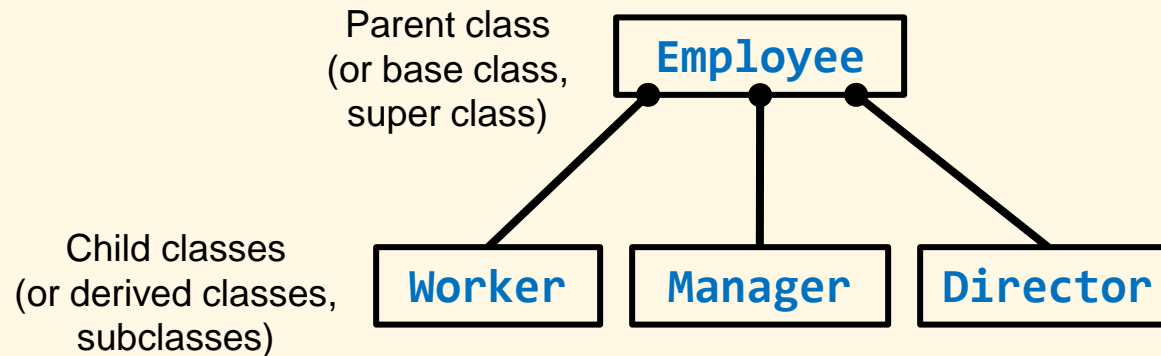
```
class Worker {  
    private String name;  
    private float salary;  
    private int level;  
  
    String getName() {...}  
    void getPaid() {...}  
    void doWork() {...}  
    ...  
}
```

```
class Manager {  
    private String name;  
    private float salary;  
    private int dept;  
  
    String getName() {...}  
    void getPaid() {...}  
    void doWork() {...}  
    ...  
}
```

```
class Director {  
    private String name;  
    private float salary;  
  
    String getName() {...}  
    void getPaid() {...}  
    void doWork() {...}  
    void assignWork() {...}  
    ...  
}
```

- ▶ All 3 above classes have common members that are the same → By defining an **Employee** class for those things and then we can:
 - ▶ Reuse code
 - ▶ Reduce written code
 - ▶ Facilitate the maintenance, modifications in the future
 - ▶ Clarify the logic in the software design

Redesigned Classes



- ▶ Child classes have access to all the members of its parent class, like if they were their own members, except the private members
- ▶ Only the parts that are different from the parent class need to be added to the child classes
- ▶ Objects of a child classes can be implicitly casted to parent class, but not inverse:

```
Employee e = new Worker(); // Ok
Worker w = new Employee(); // Error
```

```
class Employee {
    private String name;
    private float salary;

    String getName() {...}
    void getPaid() {...}
    void doWork() {...}
}

class Worker extends Employee {
    private int level;
    ...
}

class Manager extends Employee {
    private int dept;
    ...
}

class Director extends Employee {
    void assignWork() {...}
    ...
}
```


Constructors in Inheritance (1/2)

- ▶ Constructors are not inherited by child classes

```
▶ class Pet {  
    Pet() { ... }  
    Pet(String name) { ... }  
}  
  
class Dog extends Pet {  
    // ...  
}  
  
public class Hello {  
    public static void main(String argv[]) {  
        Dog d1 = new Dog();           // Ok!  
        Dog d2 = new Dog("Max");      // Error!  
    }  
}
```

Constructors in Inheritance (2/2)

- ▶ Each constructor of child class must call one of the parent's constructors, or the constructor with no parameter is implicitly used if it exists

```
▶ class Dog extends Pet {  
    Dog() {  
        super();           // call Pet(), may be omitted  
        // ...  
    }  
  
    Dog(String name) {  
        super(name);       // call Pet(name)  
        // ...  
    }  
}  
  
public class Hello {  
    public static void main(String argv[]) {  
        Dog d1 = new Dog();           // Ok!  
        Dog d2 = new Dog("Max");      // Now ok!  
    }  
}
```

Method Overriding

- ▶ A method in a child class will override the one in the parent class, if it is declared to be the same (same name, same parameters)
- ▶ The new implementation will replace the old one from the parent class for objects of the child class
- ▶ `@Override` annotation can be optionally used for automatic detection of mismatches
- ▶ Used for runtime polymorphism

```
class Pet {  
    private String name;  
  
    Pet(String name) {  
        this.name = name;  
    }  
  
    void eat(String food) {  
        System.out.printf("Pet %s is eating %s...%n", name, food);  
    }  
}  
  
class Dog extends Pet {  
    Dog(String name) {  
        super(name);  
    }  
  
    @Override  
    void eat(String food) {  
        System.out.printf("Dog %s is eating %s...%n", name, food);  
    }  
}
```

Overridden Method Test

► Let's determine the output:

► `Dog a1 = new Dog("Max");`
`a1.eat("noodle");` `// ?`

`Pet a2 = a1;`
`a2.eat("rice");` `// ?`

`Dog a3 = (Dog)a2;`
`a3.eat("bone");` `// ?`

`Pet b1 = new Pet("Daisy");`
`b1.eat("milk");` `// ?`

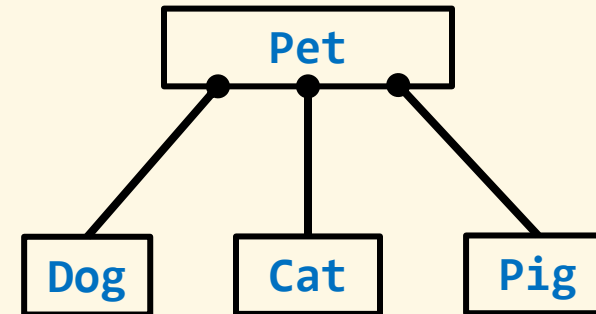
`Dog b2 = (Dog)b1;` `// ?`

Polymorphism

- ▶ Done by dynamically bind the method from the subclass in response to a method call from a subclass object referenced by superclass type
- ▶ Example: Let's add 2 more child classes **Cat** and **Pig**, then consider the following code

```
▶ Pet pets[] = {  
    new Cat("Garfield"),  
    new Dog("Max"),  
    new Cat("Lucky"),  
    new Pig("Lazy"),  
    new Dog("Quick")  
};
```

```
for (Pet p : pets)  
    p.eat("rice");
```



Methods that Cannot be Overridden

- ▶ The following methods cannot be overridden:
 - ▶ `final` methods
 - ▶ `static` methods
 - ▶ `private` methods
 - ▶ Constructors

protected Members

- ▶ The `protected` access modifier is used for members to make them accessible in the same package and subclasses
- ▶ Example: Change `private` members of Employee class to `protected`, to make them accessible by the subclasses when necessary

```
▶ class Employee {  
    protected String name;  
    protected float salary;  
  
    String getName() {...}  
    void getPaid() {...}  
    void doWork() {...}  
}
```

Object Class

- ▶ The **Object** class is the parent class of all the classes, i.e, it is the topmost class
 - ▶ No need to explicitly inherit the **Object** class
 - ▶ It provides some common behaviors to all the objects

Method	Description
<code>public final Class getClass()</code>	Returns the Class class object of this object, which can further be used to get the metadata of this class
<code>public boolean equals(Object obj)</code>	Compares the given object to this object
<code>protected Object clone()</code>	Creates and returns the exact copy (clone) of this object
<code>public String toString()</code>	Returns the string representation of this object
<code>public int hashCode()</code>	Returns the hashcode number for this object

abstract Classes

- ▶ An **abstract** method is a method declared without implementation
→ need to be overridden by the subclasses
- ▶ An **abstract** class is a class having at least one **abstract** method

- ▶ An **abstract** class cannot be instantiated

- ▶ Example:

- ▶

```
abstract class Shape {  
    abstract double getArea();  
    abstract void draw();  
    abstract void erase();
```

```
    void redraw() {  
        erase();  
        draw();  
    }
```

```
};
```

```
class Circle extends Shape {  
    // ...  
  
    @Override  
    double getArea() { //... }  
  
    @Override  
    void draw() { //... }  
  
    @Override  
    void erase() { // ... }  
}
```

Interfaces

- ▶ An interface:
 - ▶ Is a blueprint to implement a class
 - ▶ Cannot be instantiated
 - ▶ Has no concrete methods, i.e., all methods are abstract
 - ▶ Has no instance variables, but may have `public static final` ones
 - ▶ Has no constructors
- ▶ A class may inherit multiple interfaces, but can inherit only one class

▶ Example:

```
interface Drawable {  
    void draw();  
}  
  
interface Erasable {  
    void erase();  
}  
  
abstract class Shape  
implements Drawable, Erasable {  
    abstract double getArea();  
  
    void redraw() {  
        draw();  
        erase();  
    }  
}
```

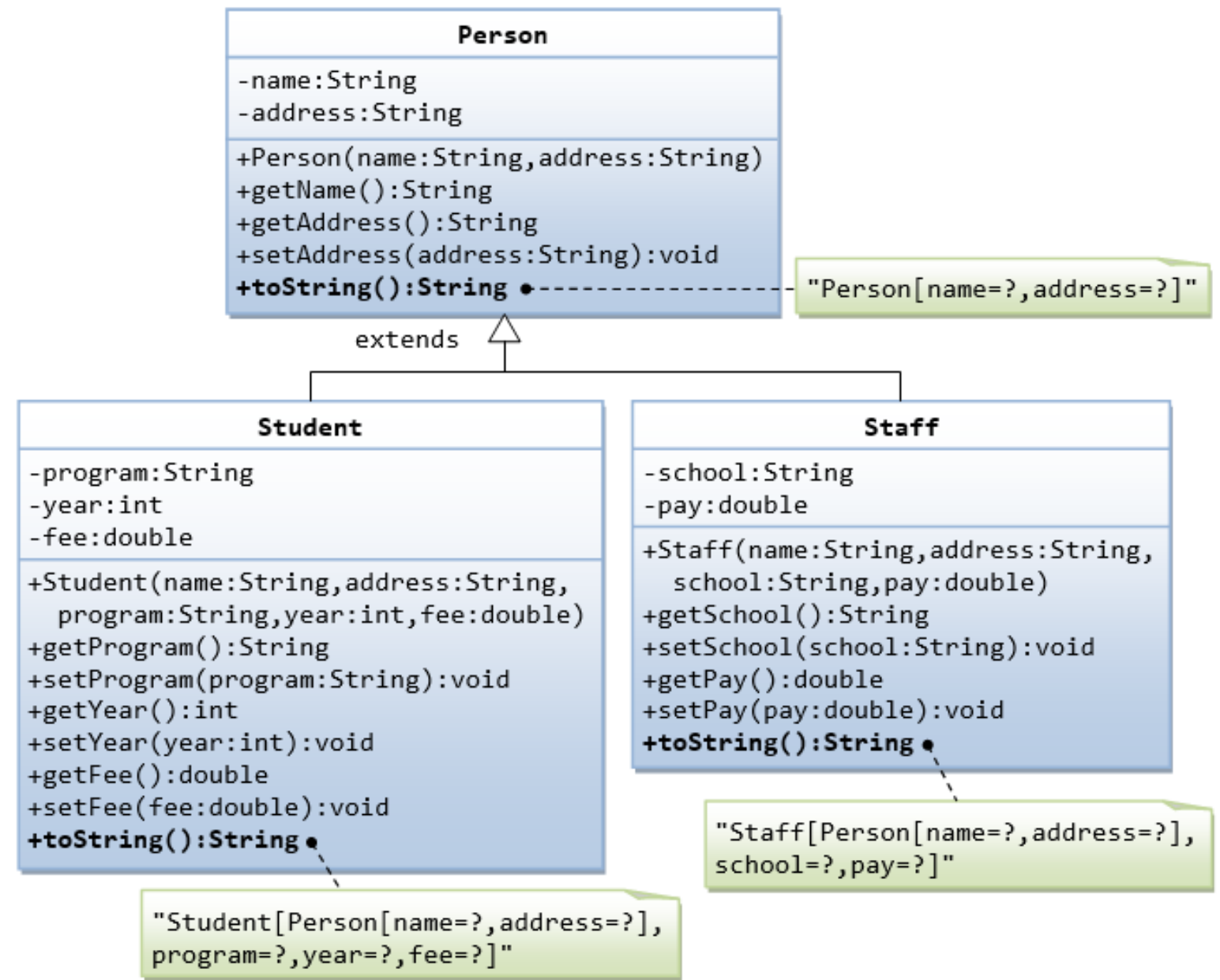
Anonymous Classes

- ▶ Anonymous classes enable programmers to declare and instantiate a class at the same time in the code
 - ▶ They are like any other classes except that don't have a name
 - ▶ Use them if one needs to use the class only once
- ▶ Example:

```
Pet aVerySpecialPet = new Pet("Unique") {  
    void sleep() { // ... }  
    void walk() { // ... }  
    void swim() { // ... }  
  
    @Override  
    void eat(String food) { // ... }  
};
```

Exercise

- Write the classes as shown in the class diagram



Packages

General

- ▶ A package is used to group related classes, and is organized as a folder in the file system
 - ▶ Better encapsulation to avoid name conflicts
 - ▶ Better maintainability
- ▶ A package may contain sub-packages → leading to a hierarchy of packages
- ▶ Categories
 - ▶ Built-in packages (from the Java API)
 - ▶ User-defined packages (your own)
 - ▶ Third-party packages (found on the Internet or given by someone else)

Built-in Packages

- ▶ The JDK provides a number of ready-to-use and useful packages

Package	Description
<code>java.lang</code>	Classes and interfaces that are fundamental to the design of the Java programming language, like <code>String</code> , <code>StringBuffer</code> , <code>System</code> , <code>Math</code> , <code>Integer</code> , ... This package is imported by default, there is no need to explicitly do that.
<code>java.util</code>	The collections framework, some internationalization support classes, properties, random number generation classes
<code>java.io</code>	Classes for system input/output operations
<code>java.awt</code>	Classes for creating user interfaces and for painting graphics and images
<code>java.net</code>	Classes for implementing networking applications
<code>java.sql</code>	Classes for accessing and processing data stored in a database

Using a Package

- ▶ Using a class without importing the package

- ▶ `java.util.Scanner in = new java.util.Scanner(System.in);`

- ▶ Import a class from a package

- ▶ `// on top of the file`
`import java.util.Scanner;`

- `// in the code`
`Scanner in = new Scanner(System.in);`

- ▶ Import a whole package

- ▶ `// on top of the file`
`import java.util.*;`

- `// in the code`
`Scanner in = new Scanner(System.in);`

User-defined Package Example

- ▶ Folders and files:

```
root
├── mypack
│   ├── MyClass1.java
│   └── MyClass2.java
```

- ▶ MyClass1.java

- ▶ `package mypack;`

- `class MyClass1 { ... }`

- ▶ MyClass2.java

- ▶ `package mypack;`

- `class MyClass2 { ... }`

Generics

General

- ▶ Sometimes, one needs to parameterize types, i.e., allow types to be parameters to methods, classes and interfaces and obtain their different versions
 - ▶ Parameters are limited to object types, it's not possible to use primitive types
- ▶ Two types:
 - ▶ Generic methods
 - ▶ Generic classes, interfaces

Generic Classes

► Example:

```
► class Pair<K, V> {  
    private K key;  
    private V value;
```

```
    Pair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }
```

```
    void setKey(K key)      { this.key = key; }  
    void setValue(V value) { this.value = value; }  
    K getKey()              { return key; }  
    V getValue()            { return value; }  
}
```

Generic Methods

► Example:

```
► class MyMath {  
    static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) &&  
            p1.getValue().equals(p2.getValue());  
    }  
}
```

► Usage:

```
► Pair<Integer, String> p1 = new Pair<>(1, "apple");  
  Pair<Integer, String> p2 = new Pair<>(2, "pear");  
  
// explicit parameters:  
boolean same1 = MyMath.<Integer, String>compare(p1, p2);  
  
// implicit parameters:  
boolean same2 = MyMath.compare(p1, p2);
```

Bounded Type Parameters

- ▶ One might want to restrict the types that can be used as type arguments in a parameterized type
 - ▶ Use the `extends` keyword, but it has a general sense to mean either `extends` or `implements`
- ▶ Examples:
 - ▶

```
class Polynomial<T extends Number> {  
    // ...  
}
```
 - ▶

```
<T extends B & A & C> void func1(T t) {  
    //...  
}
```

Exercises

1. Complete the company personnel management classes
2. Complete the class hierarchy for basic shapes: circles, rectangles, squares
3. Implement the `toString()`, `clone()`, `equals()` methods for the shape classes

