

Object Oriented Programming

(Methods in detail, Inheritance, Polymorphism)

Methods - review

- A methods are blocks of code which only run when they are **called**
- Collection of logically related instructions
- Methods are used to perform certain **actions**
- Methods are referenced by **name** – message identifier
- Data can be passed to methods – known as **parameters**
- Can have return value
- Not independent components, **parts of objects**

Method declaration

```
<visibility> <return type> <method name>([formal parameters])  
{  
    <method body>  
}
```

```
public int square(int value) {  
    return value*value;  
}
```

Method parameters

Parameters **declared** in the formal parameter list can be **referred** by their name in the message body.

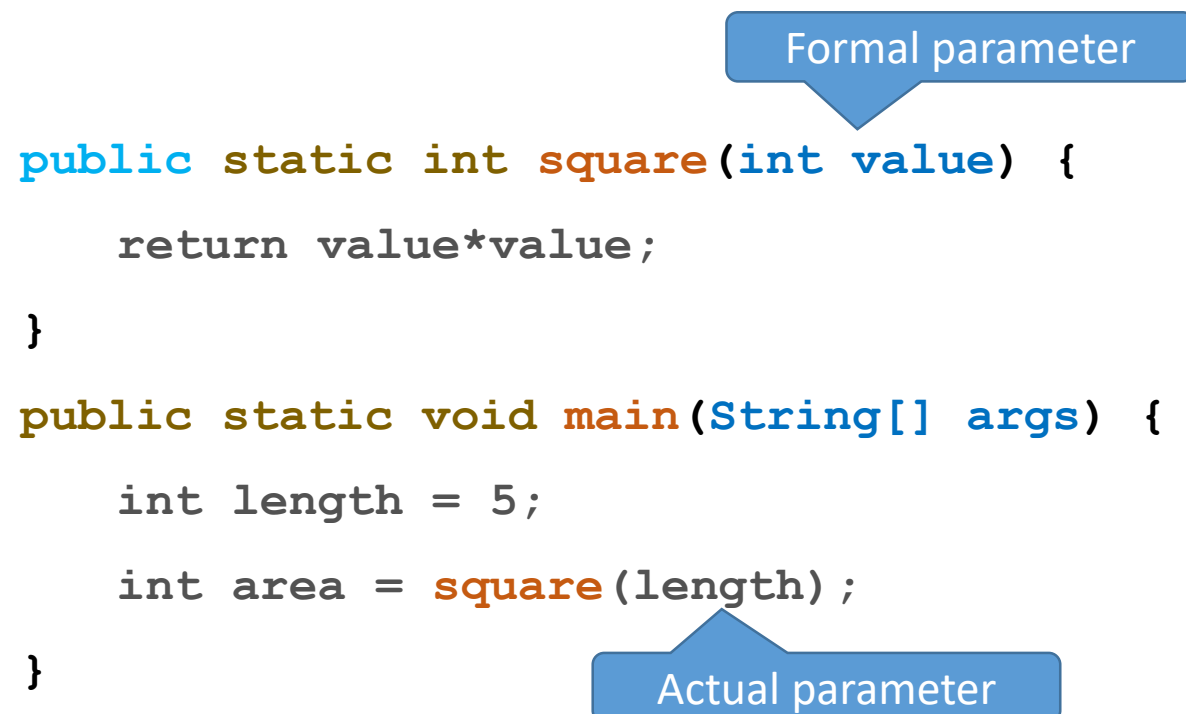
Formal parameters:

- Given in compile time

Actual parameters:

- Given in run time

```
public static int square(int value) {  
    return value*value;  
}  
  
public static void main(String[] args) {  
    int length = 5;  
    int area = square(length);  
}
```



Method parameters

- Formal parameters and actual parameters are matched in run time
- All parameters are passed by value
- Also reference type variables

Value of reference is the address of referenced region, therefore caller and method work on the same allocated area.

Method signature

Combination of

- Method name
- Method parameter type list

Not part

- Visibility modifier
- Return type
- Parameter name

```
public int calcAnswer(int op1, int op2)
{
    return op1 + op2;
}
```

Method identification

Developer calls methods by **name**, but JVM identifies by **signature**.

In different context

- Methods can exist with same name – obvious

In the same context

- Methods also can exist with same name
- Methods can NOT exist with the same signature

Method overload

An object can execute different behaviors for the same message name, because methods to execute are identified by message name and parameter types.

Defining methods in a class with same name, but different signature is called method **overload**.

Helps developers to identify **logically equal** behaviors which differ by parameter types.

Method overload

```
public int calcAnswer(int op1, int op2)
{
    return op1 + op2;
}

public float calcAnswer(float op1, float op2)
{
    return op1 + op2;
}
```

Method overload

print method of
PrintStream class

void	print (boolean b) Prints a boolean value.
void	print (char c) Prints a character.
void	print (char[] s) Prints an array of characters.
void	print (double d) Prints a double-precision floating-point number.
void	print (float f) Prints a floating-point number.
void	print (int i) Prints an integer.
void	print (long l) Prints a long integer.
void	print (Object obj) Prints an object.
void	print (String s) Prints a string.

Constructor overload

```
public Document()  
{  
    title = "";  
    author = "";  
}
```

Constructor is a method for initialization

Such a method also can be overloaded

The class could have **more constructors**

```
public Document(String title, String author)  
{  
    this.title = title;  
    this.author = author;  
}
```

Reuse I – Classes and objects

Creating more of similar things:

- Declare and object
- Create instances of objects

All objects of the same type have:

- Same state descriptors
- Same behaviors – Shared methods, working on instance data
- Custom values – instance data, instance state

Reuse II – Inheritance

- Special class **declaration** – result of object oriented design
- **Based on** an existing source class (ancestor/parent)
- New class (descendant/child) contains **all components** of the source class
 - data members
 - properties
 - behaviors
- New class can contain **new components**
- New class **extends** its ancestor

Theoretical application

Inheritance is a tool of specialization of an abstract class, to

- based on a **template** or its ancestor
- maintain more **specialized** states
described by more data members
- contain more behaviors connected to specialized functionality

Inheritance and visibility

Visibility descriptors are

- public – available from any scopes
- package private – available in declaring package, not from outside (*default*)
- protected – private + subclasses of declaring class
- private – available in scope in which component was declared

Inherited components

- Descendant class inherits **all public** states and behaviors of its ancestor
- Descendant class inherits **all protected** states and behaviors of its ancestor
- **All private** state descriptors and private behaviors also have to be inherited
- But from descendant, private components are **not available**
- Descendant inherits the **interface** of the ancestor

Inheritance in practice

```
class Document {  
    public String  
    title;  
}
```

```
class Book extends Document {  
    public int pageNumber;  
}
```

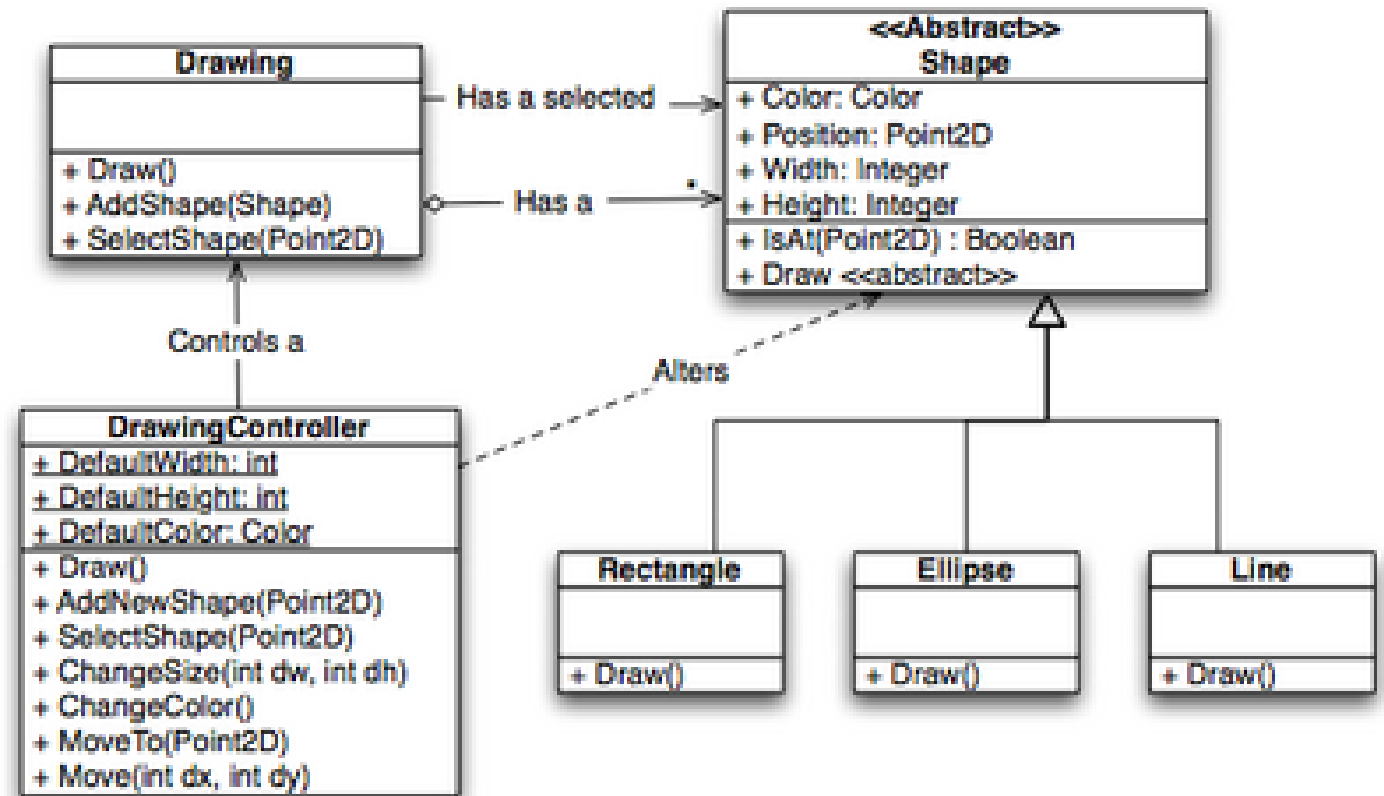
```
class FolkTale extends Book {  
    public String area;  
}
```

```
class Novel extends Book {  
    public String author;  
}
```

Class hierarchy

Can be described by

- Block diagram
- UML diagram
- ...



Single inheritance

- Each classes have **exactly one** ancestor
- If no one is specified, Object is rendered by the compiler
- The **root** is the `object` class
- Many methods are inherited by default

Methods inherited from Object

Methods	
Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Inheritance and constructor

- Constructors are **NOT** inherited
- On initialization, descendant class first calls constructor of ancestor class.
 - If other is not set explicitly, by default tries to call default constructor
 - If there is no default constructor in ancestor, descendant has to contain explicit constructor to specify what to call in ancestor

Calling constructor

```
public class Child extends Parent {  
    public Child(int someArg) {  
        super (someArg) ;  
        // custom code  
    }  
    // other extension  
}
```

Specialization of behavior

- Specialization is when declaring a descendant class based on an existing one, to handle a more special task, with **single responsibility**
- Interface of ancestor is inherited
- Ancestor extended
- *What if an existing behavior has to be specialized/modified?*

Polymorphism – method override

When modifying a behavior declared in an (direct or indirect) ancestor class .

In a lower level of abstraction, a more specialized behavior is required, therefore the descendant class declares a custom behavior, but does not modify behavior declaration of the ancestor.

The newly declared method has the same **declaration** as the old one.

Polymorphic declaration

Polymorphic method declaration components to override an inherited

- Visibility modifier can be **extended**
- Return type can **not** be changed
- Same signature
 - Name
 - Parameter types
- Parameter names can be changed

Override in practice

```
public class Mammal {  
    public String say() { return „Noise“; }  
}
```

```
public class Cat extends Mammal {  
    public String say() { return „Meow“; }  
}
```

```
public class Dog extends Mammal {  
    public String say() { return „Bark“; }  
}
```

```
public class Fox extends Mammal {  
    public String say() { return „Ring-ding-ding-ding-dingeringeding“; }  
}
```

Method override

```
public class Parent {  
    public String getClassName()  
    {  
        return "Parent";  
    }  
}
```

```
public class Child extends Parent {  
    public String getClassName()  
    {  
        return "Child";  
    }  
}
```

Call overridden methods

```
Parent parent = new Parent();
```

```
Child child = new Child();
```

```
System.out.println(parent.getClassName()); // out: Parent
```

```
System.out.println(child.getClassName()); // out: Child
```

Resolve method reference

When calling a method, reference has to be resolved

Compiler resolves the method reference by

- walking up on the class hierarchy
- using the method closest to the caller context

Polymorphism – method override

- All methods of a class could be overridden by default
- To disable method override, use `final` keyword

Plymorphism – operator override

- Java does not support operator override
(Other languages, like C# do support)
- Operators for classes can not be defined
- Exception is concatenation operator (+) on String class

Type casts

- Implicit type cast
 - Compatible type to bigger storage space
 - Compatible type to smaller storage space is **NOT ALLOWED**
 - Objects can be casted to classes higher in the hierarchy

```
        Parent parent = new Child();  
        Child child = new Parent();  
//Incompatible types
```


Implicit cast

Type cast is enabled to higher class in class hierarchy → to parent classes

Communicate with the object on an interface of its parent

For cases when inspection or collection is required on higher abstraction level

(Driving rules: vehicle – car, truck, bus, motorbyke)

Implicit cast példa

```
Vehicle[] vehicles = new Vehicle[4];  
  
vehicles[0] = new Car();  
vehicles[1] = new Truck();  
vehicles[2] = new Bus();  
vehicles[3] = new Motorbyke();  
  
foreach(Vehicle vehicle : vehicles) {  
    vehicle.accelerate();  
}
```

Explicit type casts

- Objects can be casted to any class

Type matching done runtime

```
Child child1 = new Parent();  
//Incompatible types
```

```
Child child2 = (Child) (new Parent());  
//ClassCastException
```

```
Parent parent = (Parent) new Child(); //Valid  
Child child3 = (Child) (parent);
```

Override inherited interface member

```
public class Parent {  
    public String getClassName()  
    {  
        return "Parent";  
    }  
}
```

```
public class Child extends Parent {  
    public String getClassName()  
    {  
        return "Child";  
    }  
}
```

Override inherited interface member

```
Parent parent = new Parent();    Parent parent2 = child;  
Child child = new Child();      Child child2 = (Child)parent2;
```

```
System.out.println(parent.getClassName());    // out: Parent  
System.out.println(child.getClassName());     // out: Child
```

```
System.out.println(parent2.getClassName());   // out: Child  
System.out.println(child2.getClassName());    // out: Child
```

No matter which interface the object is connected on, the called method is overridden

Static override

- Static method can **not** be overridden
- Redclaration of static method in an inherited class is allowed
- Will create a new behavior for the successor
- Will not override behavior of ancestor, just **hide** it
- Used method **depends on interfaces** used for communication

Hide static method

```
public class Parent {  
    public static String getStaticName() {  
        return "Parent";  
    }  
}  
  
    public class Child extends Parent {  
        public static String getStaticName() {  
            return "Child";  
        }  
    }
```

Hide static method

```
Child child = new Child();    Parent parent = child;    Child c2 = (Child)parent;
```

```
System.out.println(Parent.getStaticName());    // Parent
```

```
System.out.println(Child.getStaticName());    // Child
```

```
System.out.println(child.getStaticName());    // Child
```

```
System.out.println(parent2.getStaticName());    // Parent
```

```
System.out.println(child2.getStaticName());    // Child
```

The interface specifies which method will be executed

Disable inheritance – final

To exclude class from inheritance, use `final` keyword

Such a class can not be extended/subclassed

To disable method to be overridden, use `final` keyword

Such a method can not be redeclared in a descendant class