COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

2. Classes and inheritance

Aims

Introduce object oriented programming (OOP) in Java

Objectives

At the end of this lecture, you will be able to write simple classes of your own and to understand the way classes fit into inheritance hierarchies. You will also be aware of potential problems

2. Classes and inheritance

Outline

- Typical class structure
- Inheritance in Java
- Scope of fields and methods in inheritance hierarchies

Reading

Core Java (vol 1) Chapters 4 and 5

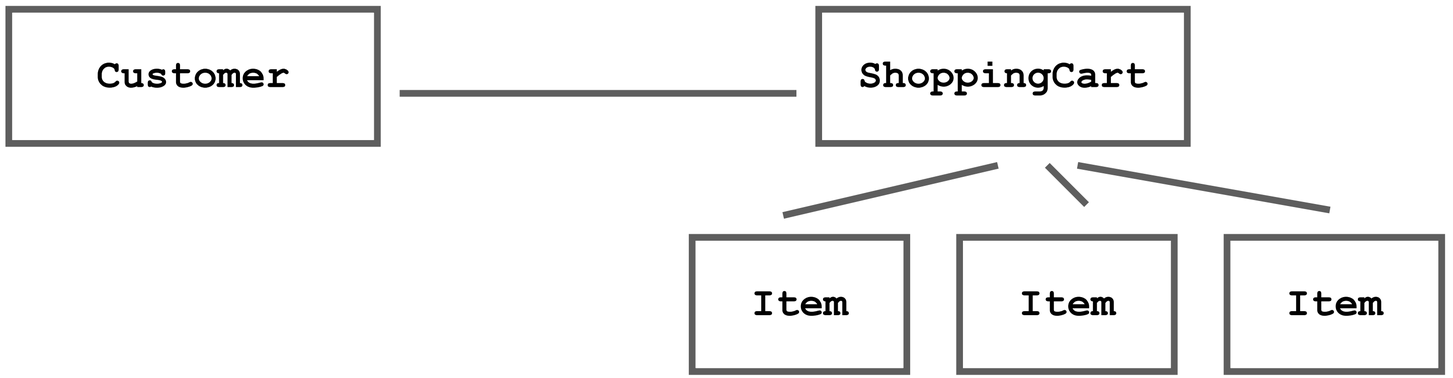
Object-oriented programming (OOP)

A Java program is made up of objects with

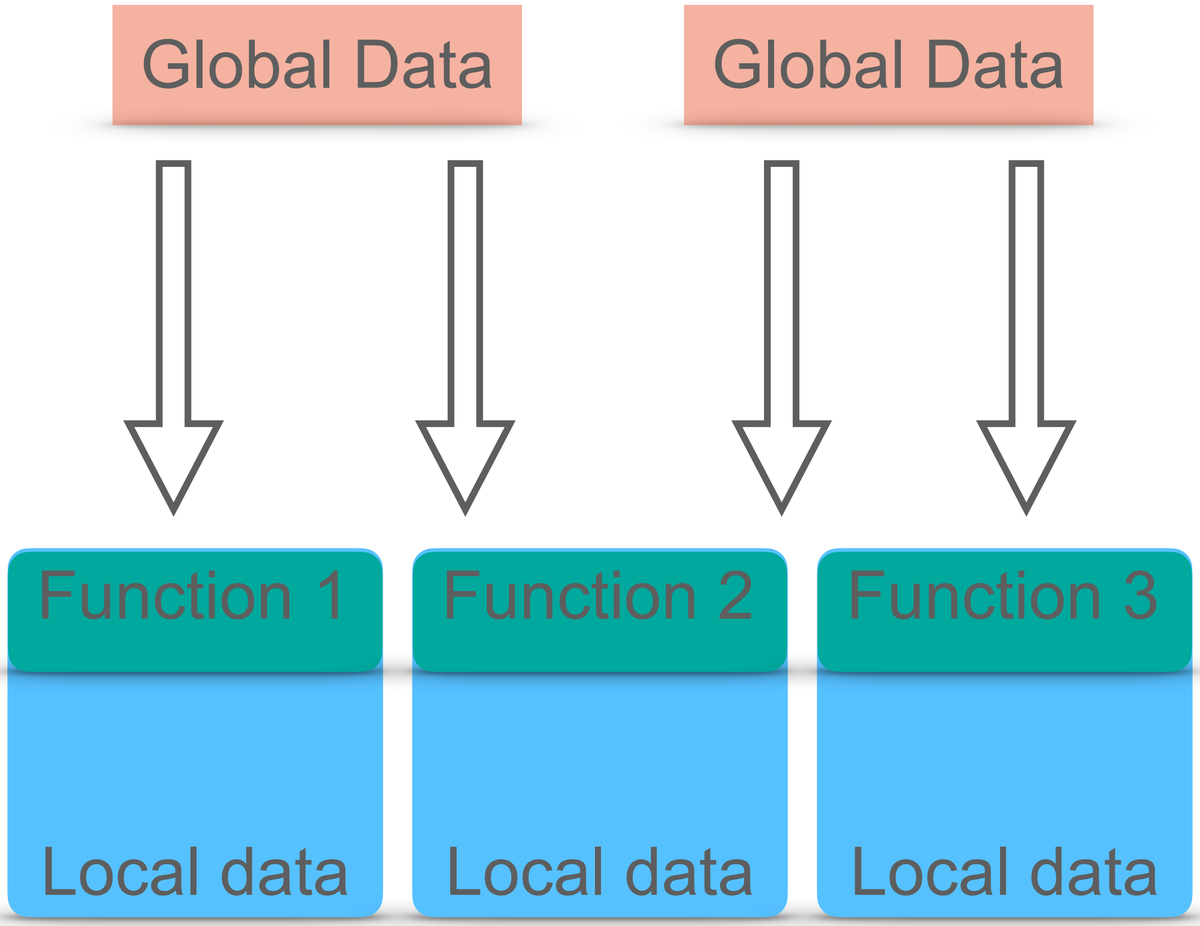
- certain properties (*fields)* and
- certain operations (*methods)*

The state of the program changes by objects interacting with each other in well-specified (and documented!) ways

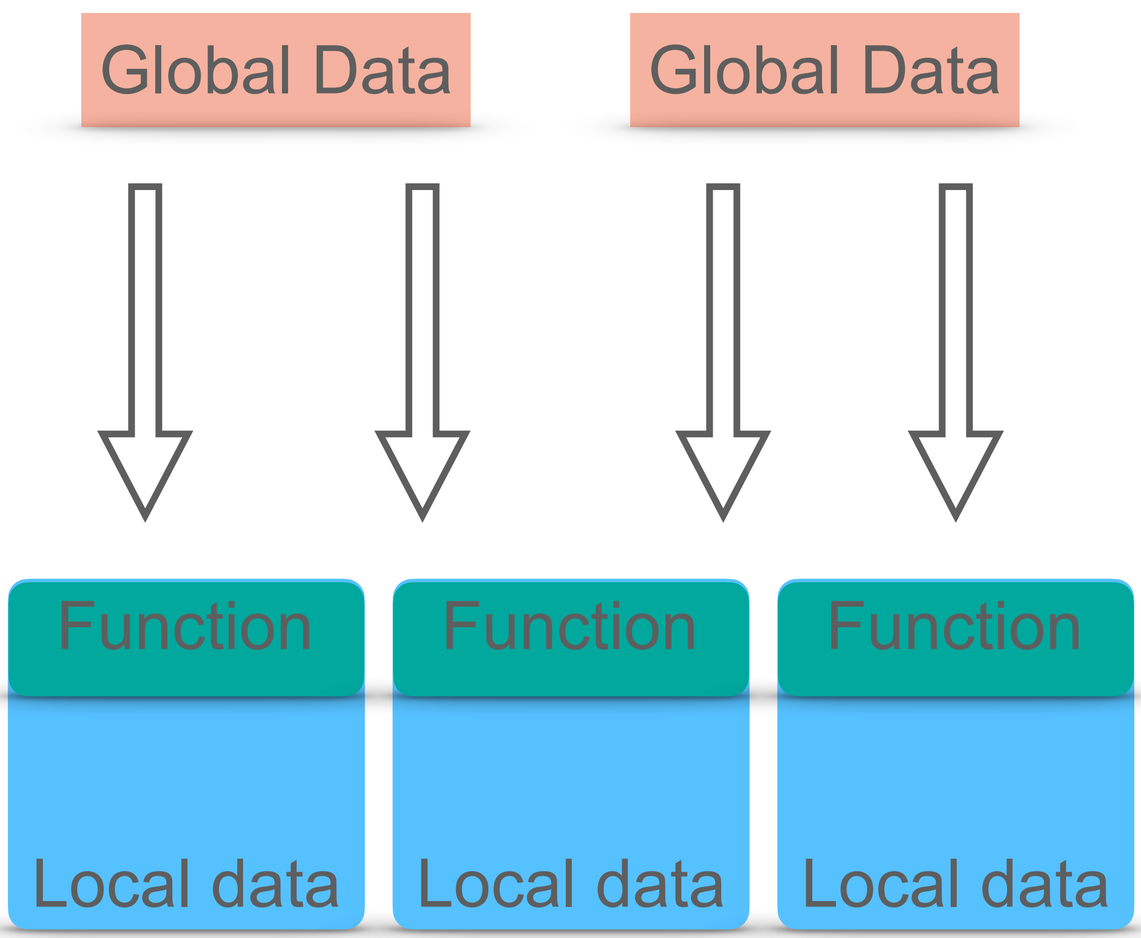
Objects are often an intuitive way to model the real world.

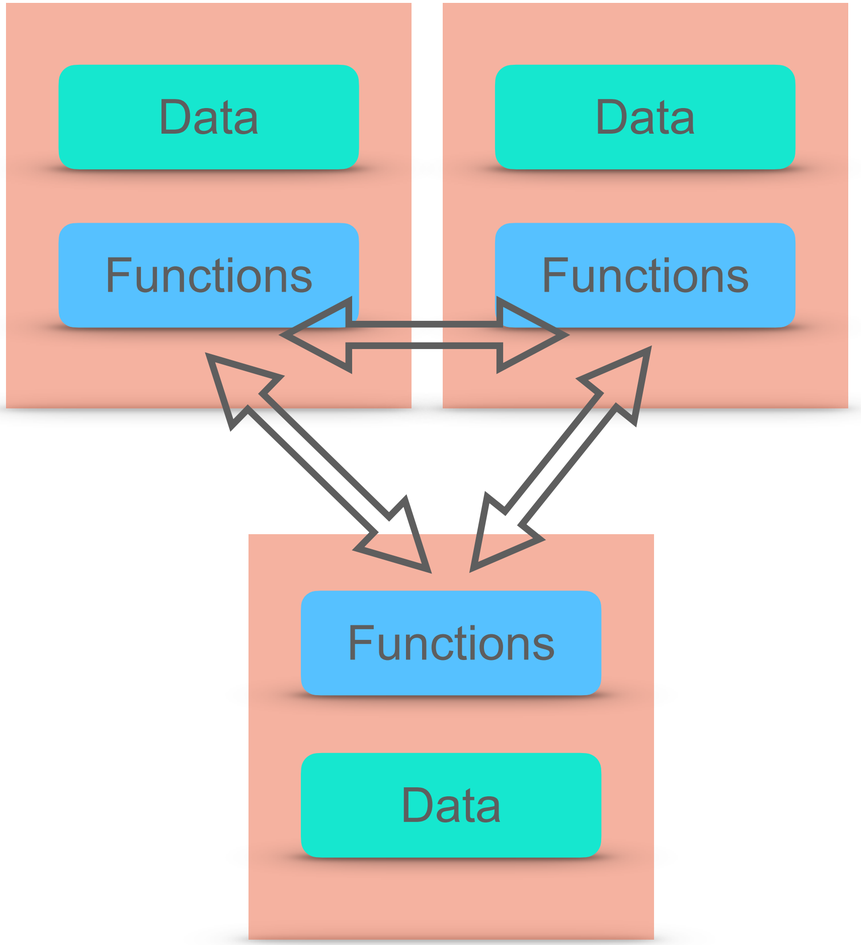


Procedural programming

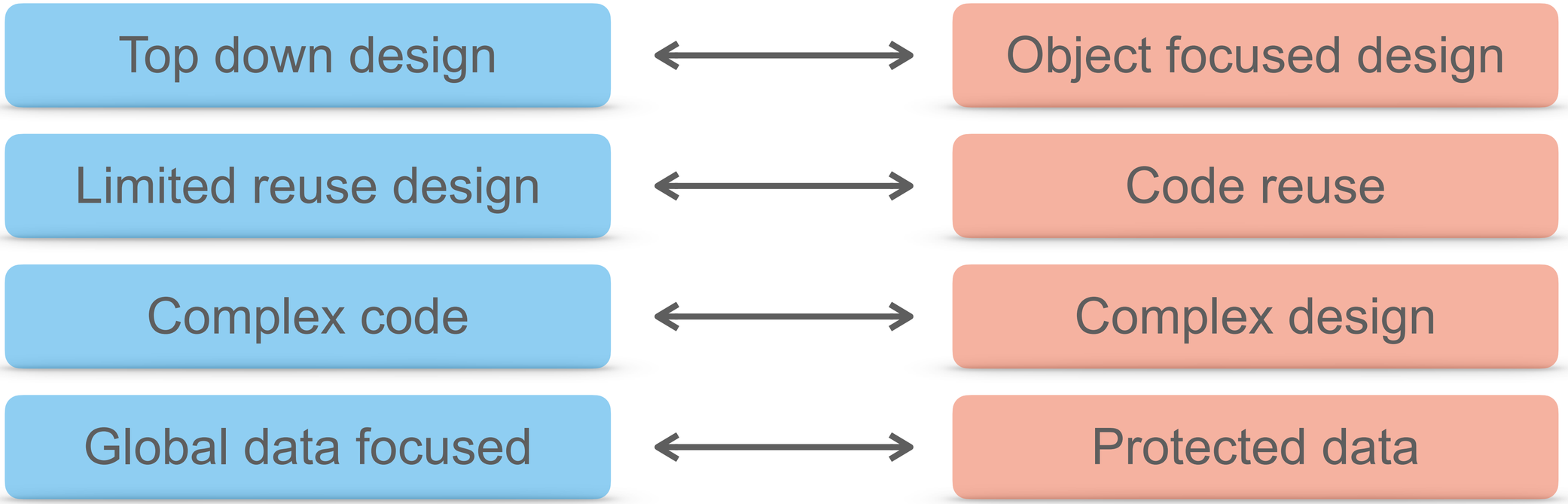


Procedural vs OOP





Procedural vs OOP



Object-oriented programming (OOP)

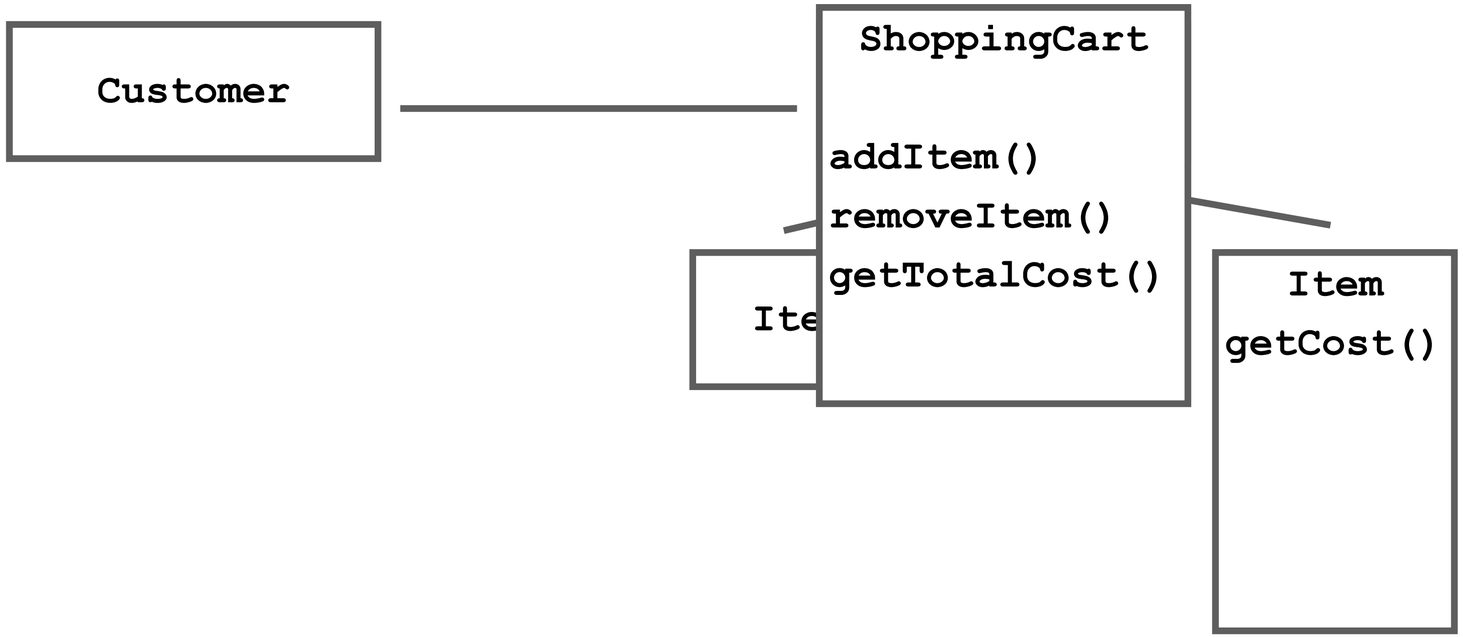
Objects are created from a *class template*

Objects of the same class store similar information and do similar things

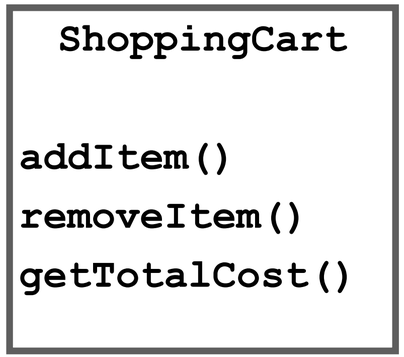
You can build your own class (programming) or use a pre-existing class

Object-oriented programming (OOP)

In OOP you care about an object's interface to the outside world, not its inner workings.



Some vocabulary



Class A class is the *blueprint for an object*, specifying the data it contains and the methods that make up its interface to the outside world

Object An object is an instance of a class, and is constructed by calling the class constructor

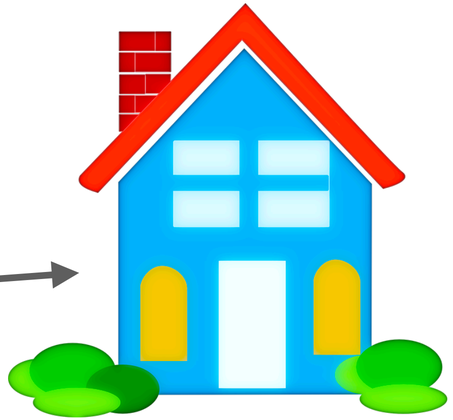
```
// some code
ShoppingCart heidisShoppingCart = new ShoppingCart ();
```

Encapsulation This means combining data and behaviour in an object, such that the implementation of the data is hidden from users of the object. They must access the object using its publicly available methods.

Class vs Object

Class

Object instances



`addChimney()`

Example — `Customer` class

```
public class Customer {









}
```

Example — `Customer` class

```
public class Customer {
    // constructor
    public Customer(String nm, String addr)
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }









}
```

Constructor

Example — `Customer` class

```
public class Customer {
    // constructor
    public Customer(String nm, String addr)
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }

    // methods
    public void addToSpend(double x) {
        totalSpend += x;
    }

   public String toString() {
        return("Customer[name=" + name + "; address=" + address + "; totalSpend=" + totalSpend + "]");
    }




}
```

Constructor

Methods

Example — `Customer` class

```
public class Customer {
    // constructor
    public Customer(String nm, String addr)
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }

    // methods
    public void addToSpend(double x) {
        totalSpend += x;
    }

   public String toString() {
        return("Customer[name=" + name + "; address=" + address + "; totalSpend=" + totalSpend + "]");
    }

    // field accessors
    public String getName() { return name; }
    public String getAddress() { return address; }
    public double getTotalSpend() { return totalSpend; }




}
```

Constructor

Methods

Field accessors

Example — `Customer` class

```
public class Customer {
    // constructor
    public Customer(String nm, String addr)
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }

    // methods
    public void addToSpend(double x) {
        totalSpend += x;
    }

   public String toString() {
        return("Customer[name=" + name + "; address=" + address + "; totalSpend=" + totalSpend + "]");
    }

    // field accessors
    public String getName() { return name; }
    public String getAddress() { return address; }
    public double getTotalSpend() { return totalSpend; }

    // instance fields
    private String name;
    private String address;
    private double totalSpend;
}
```

Constructor

Methods

Field accessors

Instance fields

Example — `Customer` class

```
public class Customer {
    // constructor
    public Customer(String nm, String addr)
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }

    // methods
    public void addToSpend(double x) {
        totalSpend += x;
    }
```

Constructor

Methods

We could have another method **`addToSpend`** with a different argument.

This is called overloading.

```
    public void addToSpend(int x) {
        totalSpend += (double) x;
    }
}
```

http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html for more details.

Some vocabulary

`Class` A class is the *blueprint for an object*, specifying the data it contains and the methods that make up its interface to the outside world

`Object` An object is an instance of a class, and is constructed by *calling the class constructor*

`Encapsulation` This means combining data and behaviour in an object, such that the implementation of the data is hidden from users of the object. They must access the object using its publicly available methods.

`Overloading` Defining multiple methods with the same name but different argument list

Overloading

```
public class Customer {
    // constructor
    public Customer(String nm, String addr)
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }

    // methods
    public void addToSpend(double x) {
        totalSpend += x;
    }

    public void addToSpend(int x) {
        totalSpend += (double) x;
    }

    // . . .

}
```

```
customer.addToSpend(3.5);
customer.addToSpend(3);


System.out.println(myString + " hello!");
System.out.println(myInt + 3);
```

The parameters' types determine which of the overloaded methods are called.

We already know this from operators, e.g. '+'

Objects are references

```
public class CustomerTest2 {
    public static void main(String[] args) {









}
```

Objects are references

```
public class CustomerTest2 {
    public static void main(String[] args) {
    // create a new customer
    Customer firstCustomer = new Customer(“H. Christensen”, "Sheffield");
}
```

When we create a new object by calling a constructor, we create a *reference* to a memory area.

Objects are references

```
public class CustomerTest2 {
    public static void main(String[] args) {
    // create a new customer
    Customer firstCustomer = new Customer("H. Christensen", "Sheffield");

    // create another new customer ?
    Customer secondCustomer = firstCustomer;




}
```

When we create a new object by calling a constructor, we create a *reference* to a memory area.

This statement copies the memory reference **not** the object

Objects are references

```
public class CustomerTest2 {
    public static void main(String[] args) {
    // create a new customer
    Customer firstCustomer = new Customer("H. Christensen", "Sheffield");

    // create another new customer ?
    Customer secondCustomer = firstCustomer;

    // use method add to customer spend
    firstCustomer.addToSpend(59.99);
    secondCustomer.addToSpend(99.99);

    // print out modified customer information
    System.out.println("First :"+firstCustomer.toString());
    System.out.println("Second:"+secondCustomer.toString()};
}
```

When we create a new object by calling a constructor, we create a *reference* to a memory area.

This statement copies the memory reference **not** the object

```
public class CustomerTest2 {
    public static void main(String[] args) {
    // create a new customer
    Customer firstCustomer = new Customer(“H. Christensen”, "Sheffield");

    // create another new customer ?
    Customer secondCustomer = firstCustomer;

    // use method add to customer spend
    firstCustomer.addToSpend(59.99);
    secondCustomer.addToSpend(99.99);

    // print out modified customer information
    System.out.println("First :"+firstCustomer.toString());
    System.out.println(“Second:”+secondCustomer.toString()};
}
```

```
>java CustomerTest2
First :Customer[name="A. Client",address="Sheffield",totalSpend=159.98]
Second:Customer[name="A. Client",address="Sheffield",totalSpend=159.98]
```

Inheritance

Inheritance is basic concept of OOP

Basic idea: create new classes that extend existing classes

Extending a class allows its methods and fields to be inherited (reused)

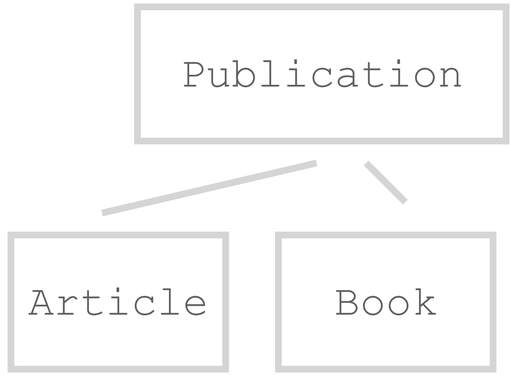
New methods and fields are added to specialise the new class

Inheritance — example

In this example magazine `Article` is a subclass of `Publication`, and `Publication` is a superclass of `Book`.

Specialisation and generalisation allow r*euse of code* because subclasses inherit functionality from their superclasses.

All publications have both similarities (title, author, isbn, number of pages) and differences (books have chapters, magazine articles have volume and issue).



Why is inheritance a good idea?

We could implement `Book` and `Article` as different Java classes.

```
public class Book{
        private int numPages;
        private int isbn;
        private String title;
        private String author;
        private int numChapters;
        public int getNumPages(){
                return numPages;
        }
}
```

```
public class MagazineArticle{
        private int numPages;
        private int isbn;
        private String title;
        private String author;
        private string magazineName;
        private int volume;
        private int issue;
        private int startPage;
        public int getNumPages(){
                return numPages;
        }
}
```

Shared information

Why is inheritance a good idea?

There is shared information, and we could use a ***superclass*** called `Publication` (sometimes called the base class or parent class), and make `Book` and `Article` ***subclasses*** (sometimes called derived classes or child classes) of this superclass.

Inheritance

```
public class Publication {
    private int numPages;
    private int isbn;
    private String title;
    private String author;
    public int getNumPages(){
        return numPages;}
    // Other methods
    }

public class Article extends Publication {
    private String magazineName;
    private int volume;
    private int issue;
    private int startPage;
    // Other methods
}

public class Book extends Publication {
    private int numChapters;
    // Other methods
}
```

Superclasses are extended using the extends keyword.

Subclasses have more functionality than superclasses.

Access modifiers

```
public class Publication {
    ...
    private String title;
    ...
// Other methods
```

Default:(`String title;`) visible within the package

Public: (`public String title;`) visible to the world, i.e. any other java class. This is usual for class methods, but breaks encapsulation for instance fields.

Private: (`private String title;`) visible within the Publication class only. This is usual for instance fields, and enables data hiding.

Protected: (`protected String title;`) visible within the package where it is declared, *and within subclasses of the class* within which it is declared

Access modifiers

Tips on Choosing an Access Level (from http://download.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html):

“If other programmers use your class, you want to ensure that errors from misuse cannot happen. Access levels can help you do this.

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.”

Constructing extended classes

If we give the subclass a constructor like this

```
public Book( String a, String t, int i, int n, int c ){
    author = a;
    title = t;
    isbn = i;
    numPages = n;
    numChapters = c;
}
```

```
public class Publication {
    private int numPages;
    private int isbn;
    private String title;
    private String author;
    public int getNumPages(){
        return numPages;
    }
}
```

Calling it with

```
Book b1 = new Book("C.Dickens", "Bleak House”,789,195,9);
System.out.println(b1.toString());
```

results in a problem because the instance fields of the `Publication` class are private, and so we cannot set `numPages`, `isbn`, `title`, and `author`.

Constructing extended classes

Possible solutions:

1. Declare instance fields of Publication to be public or something else. Bad, because this breaks encapsulation.
2. Use mutator fields (`setSomething()`) in the Publication superclass to set the instance fields.

```
// constructor
public Book( String a, String t, int i, int n, int c ){
    this.setAuthor(a);
    this.setTitle(t);
    . . . Etc
}
```

The **`this`** keyword is a reference to the current object, whose method or constructor is being called, see http://download.oracle.com/javase/tutorial/java/javaOO/thiskey.html

Constructing extended classes

A better solution is to invoke the *superclass constructor* when subclass objects are created.

```
public Book( String t, String a, int i, int n, int c ){
    super(t, a, i, n);
    this.numChapters = c;
}
```

```
public Publication( String t, String a, int i, int n ){
    this.title = t;
    this.author = a;
    this.isbn = i;
    this.numPages = n;
}
```

The `super` keyword must be the first statement in the constructor, and the parameters must exactly match the parameters of the superclass constructor.

This is an example of ***chaining constructors***, calling one constructor from within another.

Inherited methods 1

If we put a `getTitle()` method in `Publication` that has `public` or `protected` access, the `Book` subclass will inherit this method from the `Publication` superclass, and so we can use this method with `Book` objects.

```
public class PublicationTest{
   public static void main(String[ ] args){
    Publication p1 = new Publication("Richard III","W. Shakespeare",99,2 );
    Book b1 = new Book("Bleak House","C. Dickens",88, 195, 10 );
    System.out.println(p1.getTitle());
    System.out.println(b1.getTitle());
  }
}
```

```
> java PublicationTest
Richard III
Bleak House
```

What happens if we also include a `getTitle()` method in the `Book` class?

Interited methods 2

The `getTitle()` method in `Book` will *override* `getTitle()` in `Publication` but we need to take care because `title` is a `private` variable of the superclass, and we do not have an object of the superclass type to access it.

```
public class Book extends Publication {
    //constructor and other code here …
    public String getTitle(){
       return "The Book title is " + title; // will not work
    }
}
```

If we changed the scope of `title` to `public`, the code above would work, but encapsulation would be broken.

What about accessing the `getTitle()` method of the superclass?

Interited methods 3

What about accessing the `getTitle()` method of the superclass?

```
public String getTitle(){
    return "The Book title is " + getTitle(); // will not work
}
```

```
public String getTitle(){
    return "The Book title is " + super.getTitle();
}
```

Preventing inheritance

Inheritance of classes and methods can be prevented using `final`.

```
public final class Publication {
    public Publication( String t, String a, int i, int n ){
        title = t;
        author = a;
        isbn = i;
        numPages = n;
    }

    // methods etc
     }
```

No classes that extend **`Publication`** are allowed (try it!)

```
public class Publication {
    // constructor etc.

    public final String getTitle(){
        return "The Book title is " + getTitle();
    }

    // methods etc.
     }
```

The **`getTitle()`** method in **`Publication`** cannot be overridden (try it!)

Design hints for inheritance

(See Core Java end of Chapter 5)

- Place common operations and fields in the superclass.
- Don’t use protected fields unless you have a good reason – the protected mechanism does not actually provide very much protection.
- Use inheritance to model the *kind-of* relationship.
- Don’t use inheritance unless the inherited methods make sense conceptually.
- Don’t change the expected behaviour when you override a method, especially when this would complicate the overall design.

Constructor 1

The constructor is called when a new Customer object is created:

```
// constructor
public Customer(String nm, String addr) {
        name = nm;
        address = addr;
        totalSpend = 0.0;
    }
```

```
// first way to call constructor - direct
Customer firstCustomer = new Customer("H. Christensen", "Sheffield");
```

Constructor 2

It is also possible to pass a new object to a method as a parameter e.g.

```
// second method of called constructor
System.out.println(new Customer(“H. Christensen”, "Sheffield"));
String s = new Customer(“H. Christensen”, "Sheffield").toString();
```

In each of these cases the new object is created, the constructor is invoked, and the instance fields are printed to the screen or stored in the String variable s.

However, the new object is used only once, and is not stored anywhere.

Constructor 3

Constructor name is the same as the class name.

Classes can have more than one constructor with different parameters.

This allows objects to be constructed with default instance fields.

```
// constructor
public ShoppingCart (int ni, int tc) {
    numItems = ni;
    totalCost = tc;
}

public ShoppingCart(){
    numItems = 0;
    totalCost = 0;
}
```

Constructor may take zero or more parameters

Methods — syntax

modifiers type name (parameters) […] { body }

Modifier: Zero or more special keywords e.g. public (more about these later).

Type: Return type of the method, if the method returns nothing then type must be void.

Name: Identifier for the method. The same name can be allocated to more than one method if the parameter list is different – overloading.

(Parameters): Zero of more parameters given as type followed by name, separated by commas.

[…]: Exception handling – to be covered later in the module.

{ body }: The method code.

Methods — invoking

```
public void addToSpend(double x) {
        totalSpend += x;
}
```

Methods are invoked by object.method(parameters);

The following code modifies the totalSpend instance field of the firstCustomer object:

```
firstCustomer.addToSpend(59.99);
```

Methods — invoking

Special case: toSpring()

```
public String toString() {
    return("Customer[name=" + name + "]");
}
```

This code calls the `toString` method:

```
System.out.println(firstCustomer.toString());
```

This code will also work, because the java compiler automatically invokes the `toString` method when an object is referred to or concatenated with a `String`:

```
System.out.println(firstCustomer);
```

Methods — instance fields

Instance fields are declared private, and so only methods of the Customer class can access them.

```
// instance fields
private String name;
private String address;
private double totalSpend;
```

When a new object is constructed, it is allocated instance fields, and their values are initialised.

```
// field accessors
public String getName() { return name;}
public String getAddress() { return address; }
public double getTotalSpend() { return totalSpend; }
```

Methods — encapsulation

This *encapsulation* approach ensures that once instance fields are set, they can only be changed in controlled ways.

Accessor methods return instance fields, e.g.

```
public String getName() { return name; }
```

Use descriptive method names

Mutator methods manipulate instance fields, e.g.

```
public void addToSpend(double x) {
        totalSpend += x;
}
```

Another example class

```
public class Circle {
    // constructor
    public Circle( double r ){
    radius = r;
    }
    // class field
    public static final double PI = 3.1415927;
    // class method
    public static double radToDeg( double angleRad ){
        return angleRad * 180.0 / PI;
    }
    // instance methods
    public double area(){
        return PI * radius * radius;
    }
    public double circumference(){
        return 2.0 * PI * radius;
    }
    // instance field
    private double radius;
}
```

`static`: Class fields and class methods. Common to all class members

Instance methods and instance fields. Specific to each class instance

Class (static) fields

```
// class field
public static final double PI = 3.1415927;
```

public - accessible wherever the containing class is visible.

static – only one value of static field per class, however many circles we construct, there is only one copy of PI. We do not need to create an object to access it. This is how the Java Math class works.

```
// access the class field without creating an object
System.out.println("The class field PI is " + Circle.PI);
```

final – the value of the field is always set by the method constructor, and cannot be changed by class methods. By convention constants are given names in CAPITAL LETTERS.

Class (static) fields

In fact the Java Math class defines a static constant PI

```
public class Math{
   public static final double PI = 3.14159265358979323846;
```

and this can be accessed by `Math.PI`.

Without the static modifier, we would need to create an object of the Math class simply to access PI, and each Math object would have its own copy of PI.

Another example is `System.out (System.out.println())`

```
public class System{
   public static final PrintStream out = . . .
```

If this was not declared as final, we could reassign another print stream to it.

Instance fields

Instance field = Any field declared *without* the `static` modifier.

Instance fields are associated with objects, e.g., object of type `Circle` has its own copy of the field radius.

```
// instance field
private double radius;
```

Within the class definition, instance fields are referred to by name alone.

```
// instance methods
public double area(){
   return PI * radius * radius;
}
```

Method parameters

```
<implicit parameter>.<method name>(explicit parameters)
```

```
// class method
public static double radToDeg( double angleRad ){
   return angleRad * 180.0 / PI;
   }
```

Methods are invoked with statements such as, e.g. myCircle.area()

Explicit parameters appear in the method declaration, the implicit parameter does not; for instance methods the implicit parameter is an object instance.

Class (static) methods

Class methods are declared using the static keyword.
Class methods do not operate on objects, so we do not need an object of type Circle to invoke the method.

```
// access class method without creating an object
double Degrees = Circle.radToDeg( 1.5 );
System.out.println("1.5 rad is " + Degrees + " degrees");
```

When to use class (static) methods

- When you can supply all parameters explicitly.
- And/or the method only accesses class (static) fields

The Math class has many class methods because these conditions are frequently fulfilled for mathematical operations (e.g. `Math.pow`, `Math.sin`, `Math.sqrt`).

We can now understand the syntax for main methods

```
public static void main(String[ ] args) {
    . . .
}
```

Instance methods

Any method declared *without the static* modifier.

```
// instance methods
    public double area(){
      return PI * radius * radius;
      }
    public double circumference(){
      return 2.0 * PI * radius;
      }
```

Instance methods

These methods must be called with an implicit parameter that is an object, e.g.

```
Circle circ = new Circle(2.0);
double area = circ.area();
```

When the method is called, the implicit parameter can be referred to by **`this`**, so we could rewrite one of our methods as follows

```
public double area(){
   return PI * this.radius * this.radius;
}
```

In this case **`this`**`.radius` refers to the radius of the associated object.

`this` keyword

If a local variable is defined with the same name as an instance field, the `this` keyword is essential.

```
public void setRadius(double radius){
   this.radius = radius;
   }

// instance field
private double radius;
```

Dynamic binding

How does the compiler know which methods to call?

1. Method candidates: Compiler finds all possible method candidates to be called based on the class (and its superclasses) and the method name
2. Overload resolution: The compiler looks at the supplied parameters and chooses the method that matches (after type conversions) - otherwise it reports an error
3. Static binding: For a constructor, or private, static or final methods the compiler knows precisely which method to call
4. Dynamic binding: Otherwise the compiler must call the appropriate method for the type of the object to which the object variable refers. This is achieved by looking up in a method table that is precomputed by the virtual machine for each class.
5. Method table: This table lists all method signatures and the actual methods to be called.