SE2205b: Algorithms and Data Structures for Object Oriented Design

Instructor: Pirathayini Srikantha

Western University

January 9, 2019

Readings/References

- Java and Object-Oriented Design Review: Goodrich et al (Ch 1, 2)

Table of Contents

1. Java Review: Syntax and Components
2. Object-Oriented Design

Table of Contents

1. Java Review: Syntax and Components

2. Object-Oriented Design

Why Java?

- **Java** is a high-level programming language
- Composed of classes and objects
- Allows for **object-oriented** design
- Automatic **garbage collection** function
- Java vs Python: Based on objects
  - Statically typed vs dynamically typed
- Java vs C:
  - References vs pointers
  - Object-oriented vs procedural
  - Garbage collection vs memory leaks

Sample Basic Java Class

Basic syntax will be reviewed using the following sample code:

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Comments

```
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Comments can be inline (e.g. //) or block (e.g. //)**

Curly braces define a control block

Semi-colon terminates a code line

Classes and Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

A class defines a blueprint composed of attributes and methods

Objects defined under a class are composed of attributes and methods defined by the class

Components of a Class: Fields

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Fields/Instance variables hold data related to every object created via this class

Components of a Class: Constructors

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Constructors used in the creation of new objects to initialize field variables

Can have multiple constructors in a class. These are distinguished by the parameters passed

Do not have a return type

Components of a Class: Methods

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Methods used to perform actions by objects or change object state

Access Control Modifiers

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Access control modifiers: Control the level of visibility granted by the class to other entities

Public modifier: Accessed by the world

Protected modifier: Accessed by subclass(es) and the same package

Private modifier: Accessed by members of the class only

Static modifier: Associated with the class as a whole (global) rather than individual objects

Abstract modifier: Methods are not implemented (must be implemented by sub-class)

Final modifier: Like a constant (only initialized once and cannot be changed)

Declaring Variables

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Declaring instance variables:

Syntax:

[modifiers] type name1 [=initValue1], name2 [=initValue2];

Type:

boolean: true/false
char: 16-bit Unicode character
byte: 8-bit signed
short: 16-bit signed
int: 32-bit signed
long: 64-bit signed
float: 32-bit floating-point number
double: 64-bit floating-point number

Explicit typecasting:

- **Specify the type of the value**

varName1= (type) varName2
(e.g. int i = (int) j)

Implicit typecasting:

- **Allows for a widening cast and error otherwise**

(e.g.
int varName1=1;
double varName2=1.2;
varName2= varName1)

Declaring Methods

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

Declaring methods:

- **Changes the state of the object**
- **Can have methods with the same name but different parameter list**
- **Not distinguished by return types**
- **Parameters are *local* variables**

Syntax:

[modifiers] retType methodName (Type1 param1,..., TypeN paramN)
{
// Method body
}

Main Method in Java

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2);
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Main method is the starting point of execution

Can exist within a class or in a separate class (all files must be compiled together)

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2)
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

An object is dynamically allocated by invoking the constructor of the class it will belong to

Keyword new returns the reference (i.e. address) to the new object instance

Reference variable stores the address of an object instance (e.g. obj1) or null

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

?

BasicClass obj1;

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2);
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```

new BasicClass(1,2)

0X1234

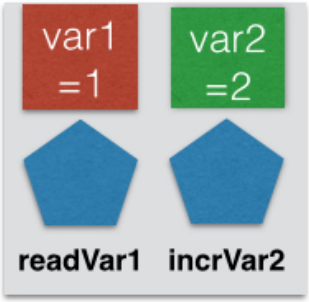
BasicClass obj1;

obj1

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2)
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```



0X1234

0X1234

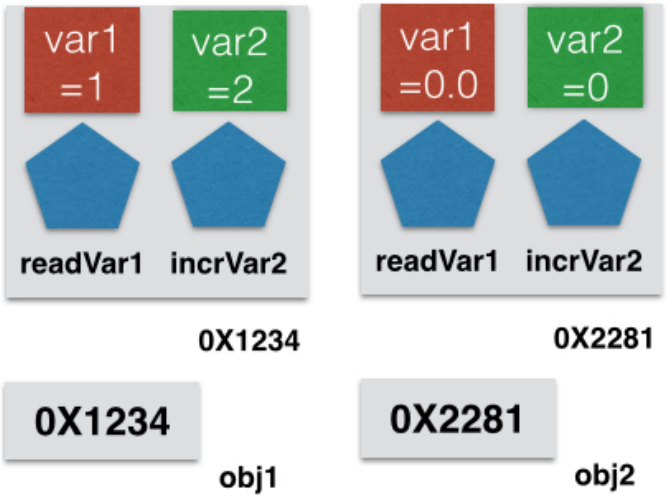
obj1

BasicClass obj1=new BasicClass(1,2);

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2)
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```



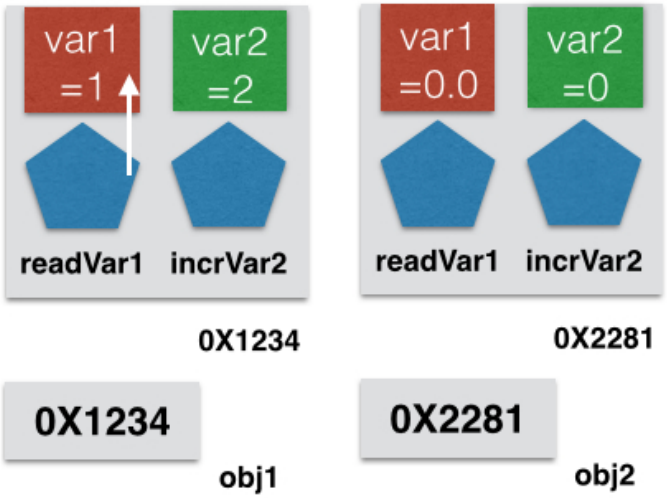
BasicClass obj1=new BasicClass(1,2);
BasicClass obj2=new BasicClass(0.0,0);

A class serve as a blueprint for objects. There can be many instances (i.e. objects) of a class.

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2);
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```



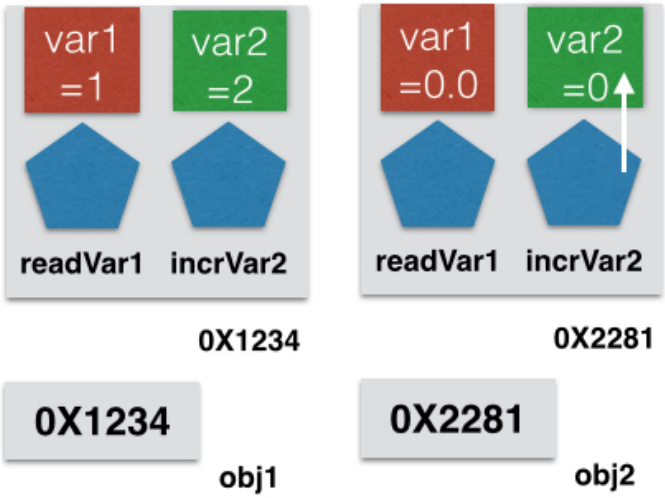
A dot operator accesses members of an object.

```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2);
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Output will be:
... 1.0

Creating Objects

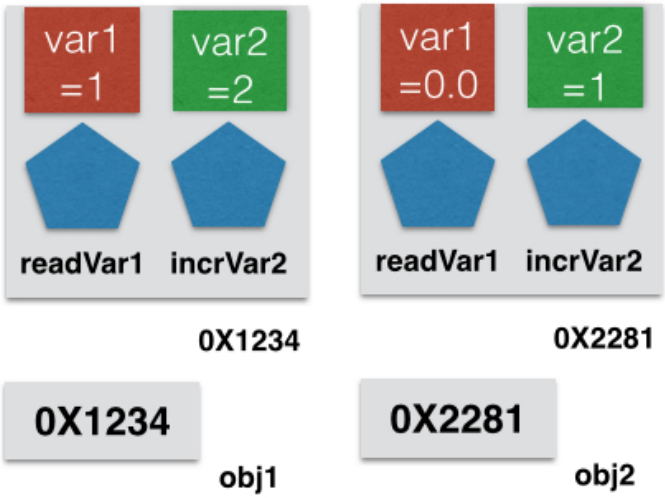
```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```




```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2);
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Creating Objects

```java
/*
 This is a demonstration of a basic Java Program
 */

public class BasicClass{
  private double var1;
  public int var2;
  public BasicClass(){
    var1=0.0;
    var2=0;
  }
  public BasicClass(double localVar1, int localVar2){
    var1=localVar1;
    var2=localVar2;
  }
  public double readVar1(){
    return var1;
  }
  public void incrVar2(){
    var2++;
  }
}
```




```java
/*
 This is a tester class
 */

public class TestClass{
  public static void main(String[] args) {
    BasicClass obj1 = new BasicClass(1,2);
    BasicClass obj2 = new BasicClass();
    System.out.println("This is the first object variable: " + obj1.readVar1());
    obj2.incrVar2();
    System.out.println("This is the second object variable: " + obj2.var2);
  }
}
```

Output will be:
... 1

`this` Keyword

- Suppose an object `obj1` belonging to `sampleClass` has been defined
- A method `sampleMethod` in `sampleClass` contains the keyword `this`
- In the context of `obj1.sampleMethod()`, `this` refers to `obj1`
- Another use of `this` is to allow an object to pass a reference to itself
- In case a local variable with the same name as another variable with a higher scope, these can be distinguished from one another
  - e.g. `public void sampleFunc(int var1){ this.var1=1;}`
- Another constructor defined in the same class can be evoked from one constructor
  - e.g. `public sampleClass(){ this.sampleClass(1);}`

Arrays

- Store ordered sequence of similar objects/variables
- Composed of n entities and indices range from 0 to $n-1$
- **Declaration of an array**:
  - Method 1: `Type[] arrayName = {value1,..., valueN};`
    e.g. `int[] sampleArray = {1,2,3};`
  - Method 2: `Type[] arrayName = new Type[n];`
    e.g. `int sampleArray[] = new int[10];`
- Accessing k^{th} element in the array: `arrayName[k]`
- If the index accessed is out of bounds, then this will result in an error (e.g. **buffer overflow** attacks)
- **Problems**: Fixed length, can run out of space

Enums

- Represent a **finite** set of **constant** integers
- Declaration of an `enum` **set**:
  - `modifier enum enumName {name1...nameN}`
    e.g. `public enum Colors {RED, YELLOW, BLUE};`
- Declaring a **variable** of type `enum`:
  - `enumName varName;`
    e.g. `Colors itemColor;`
- **Assigning** a value `nameK` to an `enum` variable:
  - `varName = enumName.nameK;`
    e.g. `itemColor = Colors.RED;`
- **More efficient** than using integer variables to denote each constant

Control Flow: If-Statements

```
public int sampleFunction(int expression){
    int retVal;
    //if-statement
    if(expression==1){
      retVal=1;
    }else if(expression==2){
      retVal=-1;
    }else{
      retVal=0;
    }
    //switch-statement
    switch(expression){
      case 1:
        retVal=1;
        break;
      case 0:
        retVal=0;
        break;
      default:
        retVal=-1;
    }
    return retVal;
}
```

If-Statement:

- Evaluates whether logical expression is true or false (==, <=, !=, etc.) from top to bottom
- Executes code block in appropriate branch

Syntax:

```
if(expression1){
...
}else if(expression2){
...
}else
...
}
```

Control Flow: Switch-Statements

```java
public int sampleFunction(int expression){
    int retVal;
    //if-statement
    if(expression==1){
      retVal=1;
    }else if(expression==2){
      retVal=-1;
    }else{
      retVal=0;
    }
    //switch-statement
    switch(expression){
      case 1:
        retVal=1;
        break;
      case 0:
        retVal=0;
        break;
      default:
        retVal=-1;
    }
    return retVal;
}
```

Switch-Statement:

- Evaluates the value of an expression from top to bottom
- Executes code block in appropriate case
- Need break to avoid executing the subsequent cases

Syntax:

```
switch(expression1){
    case Val1:
    ...
    break;
    case Val2:
    ...
    break;
    ...
    default:
}
```

Loops: For-Statements

```
public int sampleLoopFunction(int counter){
  //for-loop
  for(int i=1;i<10;i++){
    counter++;
  }

  //while-loop
  while(counter>1){
    counter--;
  }

  //do-while loop
  do{
    counter++;
  }while(counter<10);

  return counter;
}
```

for-Statement:

- **Iterates through a control block based on the conditions imposed on the parameters specified**

Syntax:

for (initialization; condition; change){

...

}

Loops: while-Statements

```
public int sampleLoopFunction(int counter){
  //for-loop
  for(int i=1;i<10;i++){
    counter++;
  }

  //while-loop
  while(counter>1){
    counter--;
  }

  //do-while loop
  do{
    counter++;
  }while(counter<10);

  return counter;
}
```

while-Statement:

- Iterates through a control block based on the condition imposed
- If a condition is false, the control block is not executed

Syntax:

```
while (initialization; condition; change){
...
}
```

Loops: Do-while-Statements

```
public int sampleLoopFunction(int counter){
  //for-loop
  for(int i=1;i<10;i++){
    counter++;
  }

  //while-loop
  while(counter>1){
    counter--;
  }

  //do-while loop
  do{
    counter++;
  }while(counter<10);

  return counter;
}
```

do-while Statement:

- **Iterates through a control block based on the condition imposed**
- **The control block is executed once even when the condition is not true**

Syntax:

while (initialization; condition; change){

...

}

Table of Contents

1 Java Review: Syntax and Components

2 Object-Oriented Design

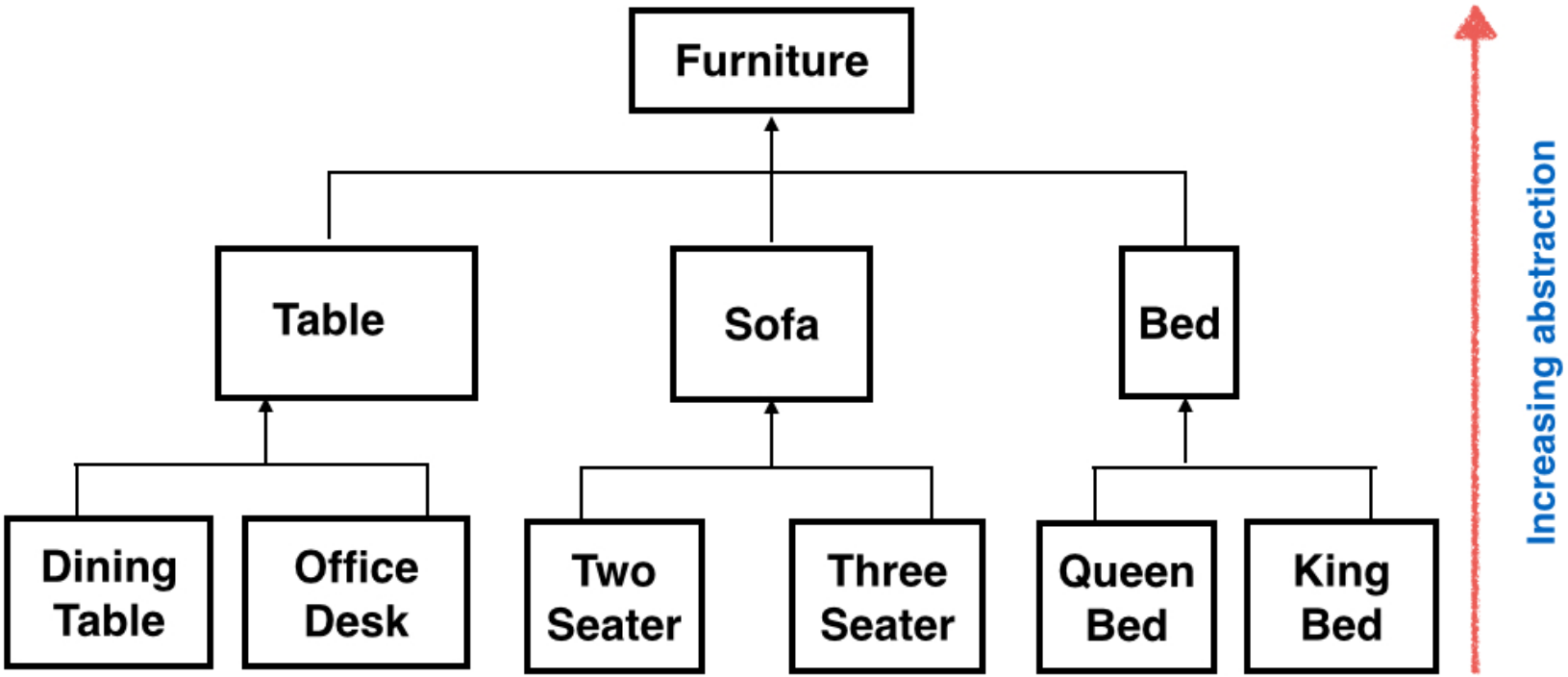
OO Design Principles

- **Abstraction**
  - **Abstract Data Types**: Data structure that hides the implementation details from users (accessed through *interfaces*)
  - Users are aware of **what** can be done by the ADT but **not how**
  - Implementation is done through *classes*
- **Encapsulation**
  - Prevents details from being revealed to reduce **dependence** on specifics of code which may change later (i.e. reusability)
  - Increases adaptability and robustness
  - Need to maintain public interface
- **Modularity**
  - Division of software program into **separate** components
  - Allows for **efficient** testing and debugging
- **Design Patterns**: Typical solutions for software problems
  - Allows for the design of OO software that is *concise*, *correct* and *reusable*

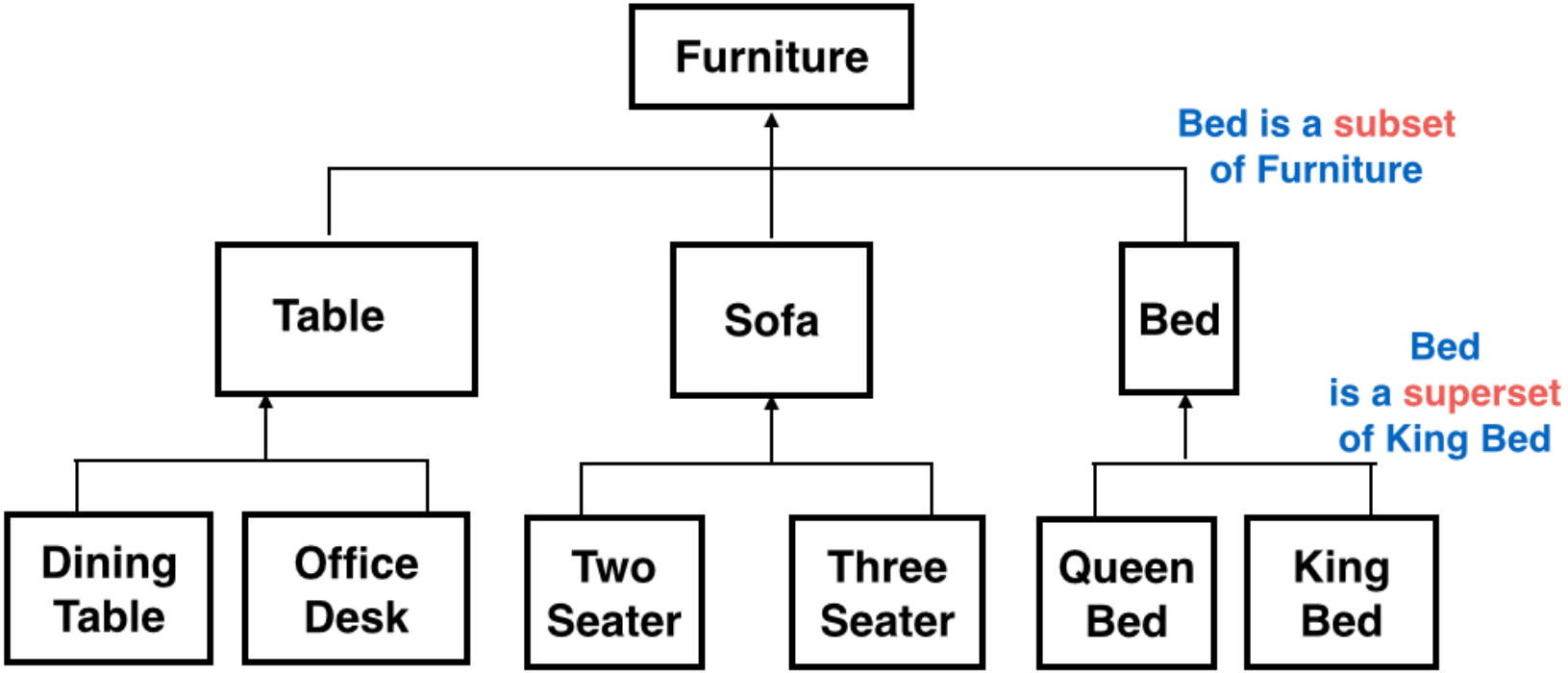
Inheritance

- **Hierarchical Design**: Group common characteristics at each level
- **Inheritance**: Allow for *modular* and hierarchical organization of code
- A new class can be defined based on an existing class
- Existing class is referred to as **base class**, **parent class**, or **super class**
- New class derived from the existing class is a **subclass** or **child** class
- Subclass **extends** the superclass
  - Inherits all methods from superclass (except for constructors)
  - Can **augment** the superclass by adding more fields/methods
  - Can **override** existing methods in superclass to make it more specific

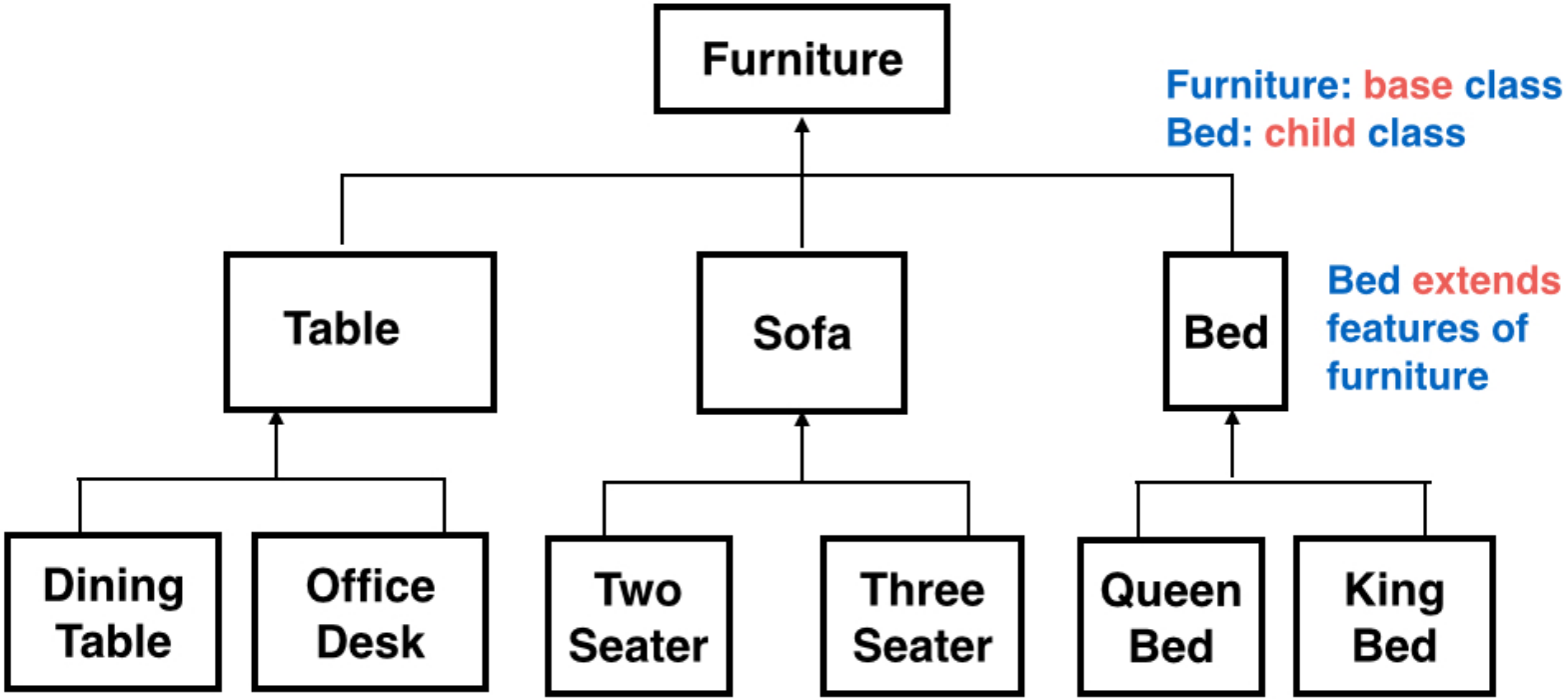
Inheritance: Example



Inheritance: Example



Inheritance: Example



Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

- 'Furniture' is the base class

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

Contains common properties of furniture

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

Contains common actions associated with furniture

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

Bed is a specific type of furniture and it extends the general class of furniture

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

New fields and methods are added/augmented in the 'Bed' subclass

These increase the specificity of the 'Bed' subclass

Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

Constructor from base class is not inherited.

super keyword is used to invoke methods and attributes from base class

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

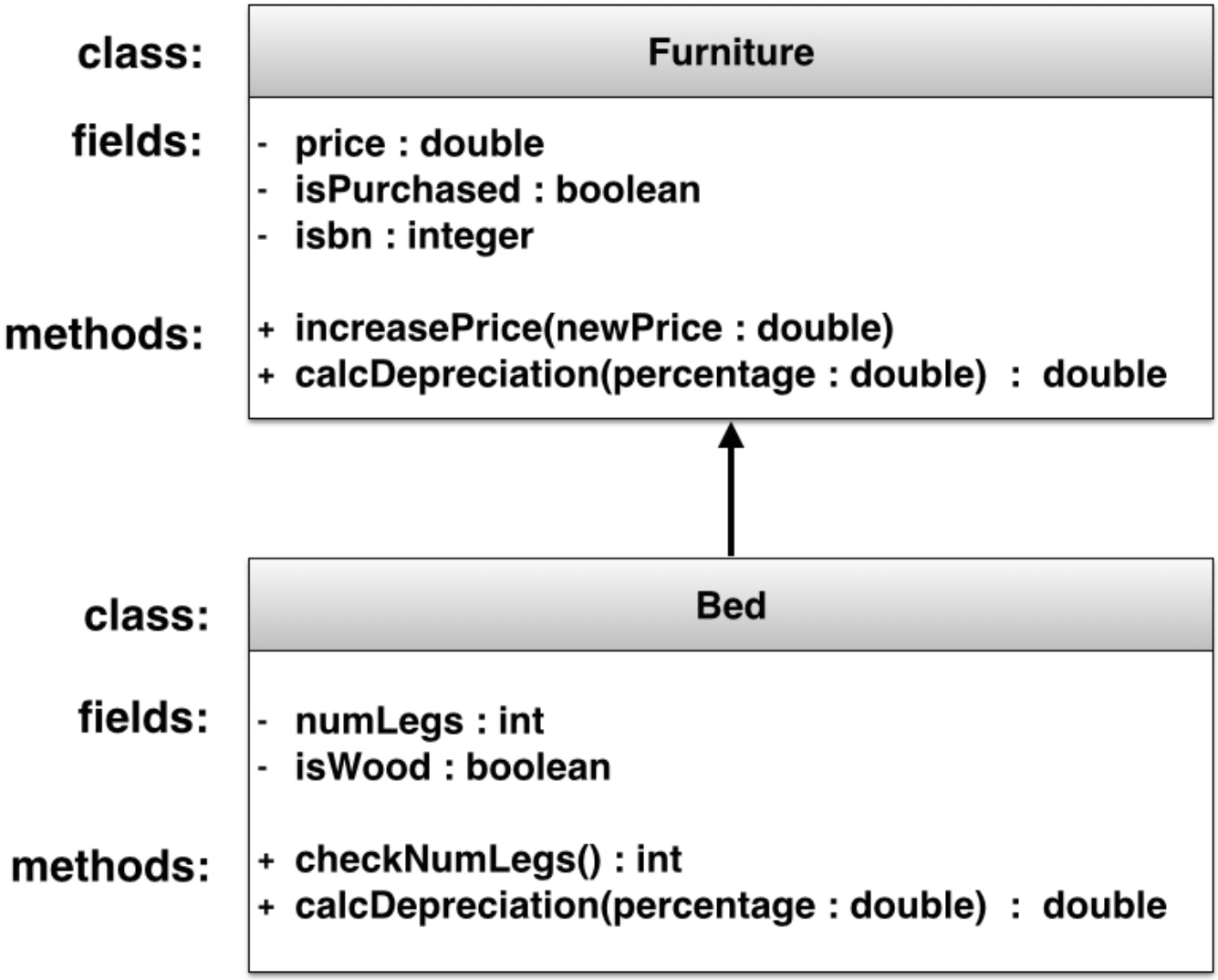
Inheritance: Code Example

```java
public class Furniture{
  private double price;
  private boolean isPurchased;
  protected int isbn;

  public Furniture(double price, int isbn, boolean state){
    this.price=price;
    this.isbn=isbn;
    isPurchased=state;
  }
  public void increasePrice(double newPrice){
    price=newPrice;
  }
  public double calcDepreciation(double percentage){
    double newPrice = price*percentage;
    return newPrice;
  }
}
```

```java
public class Bed extends Furniture{
  private int numLegs;
  private boolean isWood;

  public Bed(double price, int isbn, boolean state, int numLegs, boolean isWood){
    super(price, isbn, state);
    this.numLegs=numLegs;
    this.isWood=isWood;
  }
  public int checkNumLegs(){
    return numLegs;
  }
  public double calcDepreciation(double percentage){
    double depr =super.calcDepreciation(0.9)*0.8;
    return depr;
  }
}
```

This method is overridden to allow greater customization for the 'Bed' class

Inheritance: UML Diagram



Polymorphism

- **Polymorphism**: Ability of a reference variable to take various forms
- For instance, suppose `Furniture obj1 = new Bed();`
- Although `obj1` is of `Furniture` type, it refers to an object created by the subclass constructor
- Operator `instanceof` identifies if an instance belongs to a specific class
- Hence, `obj1` is **polymorphic**
- **Note**: It is only possible to call methods defined in the parent class (i.e. `obj1.checkNumLegs()` will result in an error)
- If the subclass has over-ridden the method, then the most specific definition is invoked via **dynamic dispatch**
- E.g. `obj1.calcDepreciation(0.9)` will call the method defined in `Bed`

Interfaces

- **Interfaces**: Type definition, methods and arguments for these
- **Strong typing**: compiler enforces conformation to the specifications of the interface
- Composed of method signatures (not constructors)
- Class **implements** one or more interface(s) according to the specified signatures
- Interfaces can inherit from other interfaces via the keyword **extends**

Interface: Code Example

```
public interface Car{
  public double drive(double direction, double speed, double duration);
  public void putBreak(boolean state);
  public void changeGears(int gearNum);
}


public class Sedan implements Car{
  private int fuelCapacity,maxSpeed, gearNum,numPersons;
  private boolean breakState;
  private double location;
  public Sedan(){
    fuelCapacity=100;
    maxSpeed=220;
    location=0;
  }
  public double drive(double direction, double speed, double duration){
    location=direction*location+speed*duration;
    return location;
  }
  public void putBreak(boolean state){
    breakState=state;
  }
  public void changeGears(int gearNum){
    this.gearNum=gearNum;
  }
  public void addPerson(){
    if(this.numPersons<5){
      this.numPersons++;
    }
  }
}
```

The interface keyword is used to define the signatures of the methods

Interface can inherit from other interfaces via the extend keyword

Interface: Code Example

```
public interface Car{
  public double drive(double direction, double speed, double duration);
  public void putBreak(boolean state);
  public void changeGears(int gearNum);
}
```

Methods signatures are defined by the inputs and outputs

Does not include variables/fields

```
public class Sedan implements Car{
  private int fuelCapacity,maxSpeed, gearNum,numPersons;
  private boolean breakState;
  private double location;
  public Sedan(){
    fuelCapacity=100;
    maxSpeed=220;
    location=0;
  }
  public double drive(double direction, double speed, double duration){
    location=direction*location+speed*duration;
    return location;
  }
  public void putBreak(boolean state){
    breakState=state;
  }
  public void changeGears(int gearNum){
    this.gearNum=gearNum;
  }
  public void addPerson(){
    if(this.numPersons<5){
      this.numPersons++;
    }
  }
}
```

Interface: Code Example

```
public interface Car{
  public double drive(double direction, double speed, double duration);
  public void putBreak(boolean state);
  public void changeGears(int gearNum);
}
```

```
public class Sedan implements Car{
  private int fuelCapacity,maxSpeed, gearNum,numPersons;
  private boolean breakState;
  private double location;
  public Sedan(){
    fuelCapacity=100;
    maxSpeed=220;
    location=0;
  }
  public double drive(double direction, double speed, double duration){
    location=direction*location+speed*duration;
    return location;
  }
  public void putBreak(boolean state){
    breakState=state;
  }
  public void changeGears(int gearNum){
    this.gearNum=gearNum;
  }
  public void addPerson(){
    if(this.numPersons<5){
      this.numPersons++;
    }
  }
}
```

A class that implements an interface must include a definition for every method signature

Can implement multiple interfaces

Interface: Code Example

```
public interface Car{
  public double drive(double direction, double speed, double duration);
  public void putBreak(boolean state);
  public void changeGears(int gearNum);
}
```

```
public class Sedan implements Car{
  private int fuelCapacity,maxSpeed, gearNum,numPersons;
  private boolean breakState;
  private double location;
  public Sedan(){
    fuelCapacity=100;
    maxSpeed=220;
    location=0;
  }
  public double drive(double direction, double speed, double duration){
    location=direction*location+speed*duration;
    return location;
  }
  public void putBreak(boolean state){
    breakState=state;
  }
  public void changeGears(int gearNum){
    this.gearNum=gearNum;
  }
  public void addPerson(){
    if(this.numPersons<5){
      this.numPersons++;
    }
  }
}
```

Additional methods can be augmented for based on the specificity of each class

Abstract Classes

- **Abstract class**: Can define method signatures which are not implemented
- Difference between interfaces and abstract classes: inclusion of variables/fields and implementation of some methods
- Non-abstract classes referred to as **concrete** class
- Abstract classes can extend another class (only single inheritance) and be extended by subclasses
- Cannot be instantiated (no constructors)

Abstract Classes: Code Example

```
public abstract class Sedan{
  private int fuelCapacity,maxSpeed, gearNum,numPersons;
  private boolean breakState;
  private double location;

  public double drive(double direction, double speed, double duration){
    location=direction*location+speed*duration;
    return location;
  }
  public void putBreak(boolean state){
    breakState=state;
  }
  public void changeGears(int gearNum){
    this.gearNum=gearNum;
  }
  public abstract void addPerson();
}
```

Keyword abstract is used in the class definition.

Abstract Classes: Code Example

```
public abstract class Sedan{
  private int fuelCapacity,maxSpeed, gearNum,numPersons;
  private boolean breakState;
  private double location;

  public double drive(double direction, double speed, double duration){
    location=direction*location+speed*duration;
    return location;
  }
  public void putBreak(boolean state){
    breakState=state;
  }
  public void changeGears(int gearNum){
    this.gearNum=gearNum;
  }
  public abstract void addPerson();
}
```

Includes methods that are not defined. Denoted by the operator abstract.

Exceptions

- **Exceptions**: Are thrown by code when an abnormal situation arises
- These can be **caught** by surrounding control blocks and handled appropriately (e.g. **try-catch**)
- If not caught, a message will be displayed in the console and the program will stop execution
- Clause **finally** executes regardless of whether an exception takes place or not
- Exceptions are thrown via: `throw new exceptionType(parameters);`
- Method declaration uses `throws` to indicate that a specific exception maybe thrown in the function

Casting

- **Widening conversion**: Casting from a narrow type to a wider type
  - No need for implicit typecasting
- **Narrowing conversion**: Casting from a wide type to narrow type (need explicit typecasting). Example:

```
Furniture objFur = new Bed();
Bed objBed = (Bed)objFur;
```

- **Casting in interfaces** occur when objects belonging to classes implementing the interface are passed as arguments to a method signature
  - Parameters are declared by the interface name instead of the class name
  - When the method is invoked, these parameters are implicitly cast into the class that implements the specific interface

Generics

- **Formal type** parameters (enclosed by angled brackets) allow for not having to specify explicit types (e.g. `public class DemoClass<A,B>`)
  - Enabled by the **generic framework** supported by Java
- `A` and `B` refer to the **generic types** which are then used to define objects and parameters as needed within the class
- When declaring an object belonging to the generic class, it is necessary to explicitly specify the type parameters (e.g. `DemoClass<Double, Integer> obj1 = new DemoClass<>(10.1,11);`)
- It is possible to make variable assignments without explicit casting
- Need to be careful with declaring arrays of objects belonging to the generic class (may get warning about casting)

Nested Classes

- A class definition can be **nested** within another class
- These are defined when the nested class is strongly related to the outer class, while increasing encapsulation and reduce conflicts in naming
- The complete name of a nested class is `Outerclass.Innerclass`
- Modifiers used to define nested class affect the visibility of the class beyond the outer class
- **Static** class has no association with any instance of the outer class (it is like a traditional class)
- Instance of a **non-static** class can only be created from within the outer class