Object-Oriented Programming
Java

Margit ANTAL
2021
Goals

1. Java Language
2. Objects and classes
3. Static Members
4. Relationships between classes
5. Inheritance and Polymorphism
6. Interfaces and Abstract Classes
7. Exceptions
8. Nested Classes
9. Threads
10. GUI Programming
11. Collections and Generics
Module 1
Java language
Java language

- History
- Java technology: JDK, JRE, JVM
- Properties
- 'Hello world' application
- Garbage Collection
Short History

- 1991 - Green Project for consumer electronics market (Oak language → Java)
- 1994 – HotJava Web browser
- 1995 – Sun announces Java
- 1996 – JDK 1.0
- 1997 – JDK 1.1 RMI, AWT, Servlets
- 1998 – Java 1.2 Reflection, Swing, Collections
- 2004 – J2SE 1.5 (Java 5) Generics, enums
- 2014 – Java SE 8 Lambdas
Short History

- 2017 - Java SE 9
- 2018 - Java SE 10, Java SE 11
- 2019 - Java SE 12, Java SE 13
- 2020 - Java SE 14, Java SE 15

Java technology

- JDK – Java Development Kit
- JRE – Java Runtime Environment
- JVM – Java Virtual Machine
Properties

- Object-oriented
- Interpreted
- Portable
- Secure and robust
- Scalable
- Multi-threaded
- Dynamic language
- Distributed
Hello World Application

1. Write the source code: HelloWorld.java

```java
public class HelloWorld{
    public static void main( String args[] ){
        System.out.println(“Hello world”);
    }
}
```

2. Compile: javac HelloWorld.java
3. Run: java HelloWorld
Hello World Application

HelloWorld.java

javac HelloWorld.java

HelloWorld.class

java HelloWorld

Runtime

JVM
Garbage Collection

- Dynamically allocated memory

- Deallocation
  - Programmer's responsibility (C/C++)
  - System responsibility (Java):
    - Is done automatically (system-level thread)
    - Checks for and frees memory no longer needed
Remember

- JVM, JRE, JDK
- Compilers vs. interpreters
- Portability
Module 2
Object-Oriented Programming
Object-oriented programming

Classes and Objects

- Class
- Attributes and methods
- Object (instance)
- Information hiding
- Encapsulation
- Constructors
- Packages
Class

- Is a **user-defined** type
  - Describes the *data* (**attributes**)
  - Defines the *behavior* (**methods**)

- Instances of a class are **objects**
Declaring Classes

- **Syntax**

  ```java
  <modifier>* class <class_name>{
    <attribute_declaration>*
    <constructor_declaration>*
    <method_declaration>*
  }
  ```

- **Example**

  ```java
  public class Counter{
    private int value;
    public void inc(){
      ++value;
    }
    public int getValue(){
      return value;
    }
  }
  ```
Declaring Attributes

• Syntax

\[
\text{<modifier>* <type> <attribute_name>[= <initial_value>];}
\]

• Examples

```java
public class Foo{
    private int x;
    private float f = 0.0;
    private String name ="Anonymous";
}
```
Declaring Methods

- Syntax

  `<modifier>* <return_type> <method_name>( <argument>* ){<statement>* }

- Examples

  ```java
  public class Counter{
      public static final int MAX = 100;
      private int value;

      public void inc(){
          if( value < MAX ){
              ++value;
          }
      }
      public int getValue(){
          return value;
      }
  }
  ```
Accessing Object Members

- Syntax
  
  `<object>}.${<member>}`

- Examples

```java
public class Counter{
    public static final int MAX = 100;
    private int value = 0;

    public void inc(){
        if( value < MAX ){
            ++value;
        }
    }

    public int getValue(){
        return value;
    }
}
```

```
Counter c = new Counter();
c.inc();
int i = c.getValue();
```
Information Hiding

- The problem:
  - Client code has direct access to internal data

```c
/* C language */
struct Date {
    int year, month, day;
};

/* C language */
Date d;
d.day = 32; //invalid day

d.month = 2; d.day = 30;
// invalid data

d.day = d.day + 1;
// no check
```
Information Hiding

- The solution:

- Client code must use setters and getters to access internal data

```java
// Java language
public class Date {
    private int year, month, day;
    public void setDay(int d){..}
    public void setMonth(int m){..}
    public void setYear(int y){..}
    public int getDay(){...}
    public int getMonth(){...}
    public int getYear(){...}
}

Date d = new Date();
//no assignment
d.setDay(32);
// month is set
d.setMonth(2);
// no assignment
d.day = 30;
```

Verify days in month
Encapsulation

- **Bundling** of **data** with the **methods** that operate on that data (restricting direct access to some of an object's components)

- Hides the **implementation details** of a class

- Forces the user to use an **interface** to access data

- Makes the code more **maintainable**
UML - Graphical Class Representation

Person

- firstName: String

+ getFirstName(): String

class name

“-” means private access

“+” means public access

State

Behaviour
Declaring Constructors

. Syntax:

```
[<modifier>]<class_name>(<argument>*){
    <statement>*
}
```

```java
public class Date {
    private int year, month, day;

    public Date( int y, int m, int d) {
        if( verify(y, m, d) ){
            year = y; month = m; day = d;
        }
    }

    private boolean verify(int y, int m, int d){
        //...
    }
}
```
Constructors

- Role: *object initialization*
- **Name** of the constructor must be the same as that of class name.
- Must **not** have **return type**.
- Every class should have **at least one constructor**.
  - If you don't write constructor, compiler will generate the **default constructor**.
- Constructors are usually declared **public**.
  - Constructor can be declared as private → You can't use it outside the class.
- One class can have **more than one constructors**.
  - Constructor **overloading**.
The Default Constructors

- There is always **at least one constructor** in every class.
- If the programmer does not supply any constructors, the default constructor is generated by the compiler
  - The default constructor takes no argument
  - The default constructor's body is empty

```java
public class Date {
    private int year, month, day;

    public Date() {
    }
}
```
Objects

- Objects are **instances** of classes
- Are **allocated on the heap** by using the new operator
- Constructor is invoked automatically on the new object

```java
Counter c = new Counter();
Date d1 = new Date(2016, 9, 23);
Person p = new Person("John","Smith");
```
Packages

- Help manage large software systems
- Contain
  - Classes
  - Sub-packages

```
java
  lang
    Math
    String
    Thread
  awt
    Graphics
    Button
```
The package statement

- Syntax:

  package <top_pkg_name>[.<sub_pkg_name>]*;

- Examples:

```java
package java.lang;

public class String{
    //...
}
```

- statement at the beginning of the source file
- only one package declaration per source file
- if no package name is declared → the class is placed into the default package
The import statement

- **Syntax:**

  \[
  \text{package } \langle \text{top_pkg_name}\rangle[.\langle \text{sub_pkg_name}\rangle]*; \\
  \]

- **Usage:**

  \[
  \text{import } \langle \text{pkg_name}\rangle[.\langle \text{sub_pkg_name}\rangle]*.*; \\
  \]

- **Examples:**

  ```java
  import java.util.List;
  import java.io.*;
  ```

  - precedes all class declarations
  - tells the compiler **where to find** classes
Remember

- Class, encapsulation
- Class members:
  - attributes
  - methods
- Object, instance
- Constructor
- Package
- Import statement
Object-oriented programming

Types

- Primitive types
- Reference Type
- Parameter Passing
- The this reference
- Variables and Scope
- Casting
Java Types

- **Primitive (8)**
  - **Logical:** boolean
  - **Textual:** char
  - **Integral:** byte, short, int, long
  - **Floating:** double, float

- **Reference**
  - All others
Logical - boolean

- Characteristics:
  - Literals:
    - true
    - false
  - Examples:
    - boolean cont = true;
    - boolean exists = false;
- Characteristics:
  - Represents a 16-bit Unicode character
  - Literals are enclosed in single quotes (' ')
  - Examples:
    - 'a' - the letter a
    - '\t' - the TAB character
    - '\u0041' - a specific Unicode character ('A') represented by 4 hexadecimal digits
Integral – byte, short, int, and long

- Characteristics:
  - Use three forms:
    - Decimal: 67
    - Octal: 0103 \((1 \times 8^2 + 0 \times 8^1 + 3 \times 8^0)\)
    - Hexadecimal: 0x43
  - Default type of literal is int.
  - Literals with the L or l suffix are of type long.
## Integral – byte, short, int, and long

- **Ranges:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>$-2^7..2^7-1$</td>
</tr>
<tr>
<td>short</td>
<td>2 byte</td>
<td>$-2^{15}..2^{15}-1$</td>
</tr>
<tr>
<td>int</td>
<td>4 byte</td>
<td>$-2^{31}..2^{31}-1$</td>
</tr>
<tr>
<td>long</td>
<td>8 byte</td>
<td>$-2^{63}..2^{63}-1$</td>
</tr>
</tbody>
</table>
Floating Point – float and double

- Characteristics:
  
  ● Size:
    - float  – 4 byte
    - double – 8 byte
  
  ● Decimal point
    - 9.65  (double, default type)
    - 9.65f or 9.65F (float)
    - 9.65D or 9.65d (double)
  
  ● Exponential notation
    - 3.41E20  (double)
public class MyDate{
    private int day = 26;
    private int month = 9;
    private int year = 2016;

    public MyDate( int day, int month, int year){
        ...
    }
}

MyDate date1 = new MyDate(20, 6, 2000);
Constructing and Initializing Objects

```java
MyDate date1 = new MyDate(20, 6, 2000);
```
Constructing and Initializing Objects

1) Memory is allocated for the object
2) Explicit attribute initialization is performed
3) A constructor is executed
4) The object reference is returned by the new operator

```java
MyDate date1 = new MyDate(20, 6, 2000);

new MyDate(20, 6, 2000);
```
Constructing and Initializing Objects

1) Memory is allocated for the object
2) Explicit attribute initialization is performed
3) A constructor is executed
4) The object reference is returned by the new operator
5) The reference is assigned to a variable

```
MyDate date1 = new MyDate(20, 6, 2000);
```
(1) Memory is allocated for the object

```java
MyDate date1 = new MyDate(20, 6, 2000);
```

**Reference**
- `date1`  

**Object**
- `day`  | 0
- `month` | 0
- `year`  | 0

Implicit initialization
(2) Explicit Attribute Initialization

```java
public class MyDate {
    private int day = 26;
    private int month = 9;
    private int year = 2016;
}
```

```
MyDate date1 = new MyDate(20, 6, 2000);
```

<table>
<thead>
<tr>
<th>Object</th>
<th>day</th>
<th>Month</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>26</td>
<td>9</td>
<td>2016</td>
</tr>
</tbody>
</table>

Reference:
- date1
- ???
(3) Executing the constructor

```java
MyDate date1 = new MyDate(20, 6, 2000);
```

<table>
<thead>
<tr>
<th>reference</th>
<th>date1</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>day</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>month</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>year</td>
<td>2000</td>
</tr>
</tbody>
</table>

```java
public class MyDate {
    private int day = 26;
    private int month = 9;
    private int year = 2016;
}
```
(4) The object reference is returned

```java
MyDate date1 = new MyDate(20, 6, 2000);
```

<table>
<thead>
<tr>
<th>reference</th>
<th>date1</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>day</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>month</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>year</td>
<td>2000</td>
</tr>
</tbody>
</table>

The address of the object: 0x01a2345
The reference is assigned to a variable

MyDate date1 = new MyDate(20, 6, 2000);

The reference points to the object
Assigning References

- Two variables refer to a single object

```java
MyDate date1 = new MyDate(20, 6, 2000);
MyDate date2 = date1;
```

![Diagram of the relationship between date1 and date2, showing that they reference the same object with the same values for day, month, and year.](image)
public class PassTest{
    public void changePrimitive(int value){
        ++value;
    }

    public void changeReference(MyDate from, MyDate to){
        from = to;
    }

    public void changeObjectDay(MyDate date, int day){
        date.setDay( day );
    }
}
Parameter Passing

Pass-by-Value

```java
PassTest pt = new PassTest();
int x = 100;
pt.changePrimitive( x );
System.out.println( x );

MyDate oneDate = new MyDate(3, 10, 2016);
MyDate anotherDate = new MyDate(3, 10, 2001);

pt.changeReference( oneDate, anotherDate );
System.out.println( oneDate.getYear() );

pt.changeObjectDay( oneDate, 12 );
System.out.println( oneDate.getDay() );
```

Output:

100
2016
12
The this Reference

- **Usage:**
  - To resolve **ambiguity** between *instance variables* and *parameters*
  - To **pass** the current *object as a parameter* to another method
public class MyDate{
    private int day = 26;
    private int month = 9;
    private int year = 2016;
    public MyDate( int day, int month, int year){
        this.day = day;
        this.month = month;
        this.year = year;
    }
    public MyDate( MyDate date){
        this.day = date.day;
        this.month = date.month;
        this.year = date.year;
    }
    public MyDate createNextDate(int moreDays){
        MyDate newDate = new MyDate(this);
        //... add moreDays
        return newDate;
    }
}
Java Coding Conventions

- **Packages**
  - ro.sapientia.ms

- **Classes**
  - SavingsAccount

- **Methods**
  - getAmount()

- **Variables**
  - amount

- **Constants**
  - NUM_CLIENTS
Variables and Scope

- Local variables are
  - Defined inside a method
  - Created when the method is executed and destroyed when the method is exited
  - Not initialized automatically
  - Created on the execution stack
public class ScopeExample {
    private int i=1;

    public void firstMethod() {
        int i=4, j=5;
        this.i = i + j;
        secondMethod(7);
    }
    public void secondMethod(int i) {
        int j=8;
        this.i = i + j;
    }
}

public class TestScoping {
    public static void main(String[] args) {
        ScopeExample scope = new ScopeExample();

        scope.firstMethod();
    }
}
### Default Initialization

- Default values for attributes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>float</td>
<td>0.0f</td>
</tr>
<tr>
<td>double</td>
<td>0.0d</td>
</tr>
<tr>
<td>char</td>
<td>'\u0000'</td>
</tr>
<tr>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td>reference</td>
<td>null</td>
</tr>
</tbody>
</table>
Operators

- Logical operators
- Bitwise operators ( ~, ^, &, |, >>, >>>, <<< )
- String concatenation ( + )
String Types

- **String**
  - Immutable – once created can not be changed
  - Objects are stored in the Constant String Pool

- **StringBuffer**
  - Mutable – one can change the value of the object
  - Thread-safe

- **StringBuilder**
  - The same as StringBuffer
  - Not thread-safe
Object-oriented programming

Arrays

- Declaring arrays
- Creating arrays
- Arrays of primitive and reference type
- Initialization of elements
- Multidimensional arrays
Declaring Arrays

- What is an array?
  - Group of data objects of the same type
- Arrays of primitive types:
  ```java
  int t[];
  int [] t;
  ```
- Arrays of reference types:
  ```java
  Point p[];
  Point[] p;
  ```
Creating Arrays

Primitive Type

- Arrays are **objects** → are created with **new**
- Example:

```java
//array declaration
int [] t;

//array creation
t = new int[10];

//print the array - enhanced for loop
for( int v: t ){
    System.out.println( v );
}
```
Creating Arrays

Primitive Type

//array declaration
int [] t;

//array creation
t = new int[10];
Creating Arrays

Reference Type

- Example:

```java
//array declaration
Point [] t;

//array creation - array of references!!!
t = new Point[3];

// How many objects of type Point?
```
Creating Arrays

Reference Type

- Example:

```java
//array declaration
Point [] p;

//array creation - array of references!!!
p = new Point[3];

// How many objects of type Point?
for (int i=0; i<3; ++i){
    p[i] = new Point(i, i);
}
// How many objects of type Point?
```
Creating Arrays

Reference Type

Stack

Heap Memory

Point []

p

:Point
(0,0)

:Point
(1,1)

:Point
(2,2)
Initialize Arrays

- Create an array with initial values

  ```java
  String names[] = {"Anna", "Krisztina", "Rebekka"};
  Point points[] = { new Point(0,0), new Point(1,1)};
  ```
Array Bounds

```java
void printElements( int t[] ){
    for( int i=0; i < t.length; ++i){
        System.out.println( t[i] );
    }
}
```
Multidimensional Arrays

- **Rectangular** arrays:
  
  ```java
  int [][] array = new int[3][4];
  ```

- **Non-rectangular** arrays:
  
  ```java
  int [][] array;
  array = new int[2][];
  array[0] = new int[3];
  array[1] = new int[5];
  ```
Remember

- Array declaration and creation
  - Array of primitives
  - Array of references
- Size of an array (public attribute: length)
- Initial values of array elements
Module 3
Static Members
Problems

- How can you create a constant?
- How can you declare data that is shared by all instances of a given class?
- How can you prevent a class from being subclassed?
- How can you prevent a method from being overridden?
Create a `Product` class which initializes each new instance with a `serialNumber` (1, 2, 3,...)
public class Product{
    private int sNumber;
    public static int counter = 0;
    public Product() {
        counter++;
        sNumber = counter;
    }
}
Product p1 = new Product();
Product p2 = new Product();
What's wrong?

```java
public class Product{
    private int sNumber;
    public static int counter = 0;
    public Product() {
        counter++;
        sNumber = counter;
    }
}

It can be accessed from outside the class!

public class AnyClass{
    public void increment() {
        Product.counter++;
    }
}
```
public class Product{
    private int sNumber;

    private static int counter = 0;

    public static int getCounter(){
        return counter;
    }

    public Product() {
        counter++;
        sNumber = counter;
    }
}
public class Product {
    private int sNumber;

    private static int counter = 0;

    public static int getCounter() {
        return counter;
    }

    public Product() {
        counter++;
        sNumber = counter;
    }
}

System.out.println(Product.getCounter());
Product p = new Product();
System.out.println(Product.getCounter());

Output?
Accessing static members

Recommended:

```
<class name>.<member_name>
```

Not recommended (but working):

```
<instance_reference>.<member_name>
```

```java
System.out.println(Product.getCounter());
Product p = new Product();
System.out.println(p.getCounter());
```

Output?
Static Members

- Static data + static methods = static members

- Data are allocated at class load time → can be used without instances

- Instance methods may use static data. Why?

- Static methods cannot use instance data. Why?
The `InstanceCounter` class

```java
class InstanceCounter {
    private static int counter;

    public InstanceCounter() {
        ++counter;
    }

    public static int getCounter() {
        return counter;
    }
}
```

```java
System.out.println( InstanceCounter.getCounter());
InstanceCounter ic = new InstanceCounter();
System.out.println( InstanceCounter.getCounter());
```
public class Singleton {
    private static Singleton instance;

    private Singleton()
    {
    
    }

    public static Singleton getInstance()
    {
        if( instance == null )
        {
            instance = new Singleton();
        }
        return instance;
    }
}
public class AClass{

    private static int counter;

    static {
        // e.g. read counter from a file
    }
}
The `final` Keyword

- **Class**
  - You cannot subclass a `final` class.

- **Method**
  - You cannot override a `final` method.

- **Variable**
  - A `final` variable is a constant.
  - You can set a `final` variable only once.
  - Assignment can occur independently of the declaration (`blank final variable`).
public class Employee{
    private final long ID;

    public Employee(){
        ID = createID();
    }

    private long createID(){
        //return the generated ID
    }
    ...
}
public enum GestureType {
    UP,
    RIGHT,
    DOWN,
    LEFT
}

for(GestureType type: GestureType.values()){
    System.out.println( type );
}

OUTPUT:
UP
RIGHT
DOWN
LEFT
public enum GestureType {
    UP (0, "fel"),
    RIGHT (1, "jobb"),
    DOWN (2, "le"),
    LEFT (3, "bal");

    GestureType( int value, String name ){
        this.value = value;
        this.name = name;
    }

    public int getValue(){
        return value;
    }

    public String getName(){
        return name;
    }

    private int value;
    private String name;
}
Enumerations

```java
for(GestureType type: GestureType.values()){  
    System.out.println(type.name()+", "+ 
                         type.getName()+", "+ type.getValue());  
}
```

Output

```
UP, fel, 0
RIGHT, jobb, 1
DOWN, le, 2
LEFT, bal, 3
```
REMEMBER

- **Constant instance data**
  - belongs to the instance

- **Static data**
  - belongs to the class

- **Constant static data**
  - belongs to the class
public class Product{
    private final int ID;
}

REMEMBER
CONSTANT INSTANCE DATA

final
REMEMBER
STATIC DATA

```
static

public class Product{
    private final int ID;
    private static counter;
    public Product(){
        ID = ++counter;
    }
}
```
public class Product{
    private final int ID;
    private static counter;
    private static final String name = "PRODUCT";
    public Product(){
        ID = ++counter;
    }

    public String getIDStr(){
        return name+ID;
    }
}

Module 4
Relationships between classes
Object-oriented programming
Relationships between classes

- **Association** (containment)
  - Strong – **Composition**
  - Weak – **Aggregation**
Relationships between classes

Composition

- Strong type of association
- Full ownership
Relationships between classes

Aggregation

- Weak type of association
- Partial ownership
Relationships between classes

Association – Aggregation - Composition
Relationships between classes
Implementing Associations (1)

```java
public class Brain{
    //...
}

public class Person{
    private Brain brain;
    //...
}
```
Relationships between classes
Implementing Associations (2)

```java
class Date{
    private int day;
    private int month;
    private int year;
    //...
}

class Person{
    private String name;
    private Date birthDate;

    public Person(String name, Date birthDate){
        this.name = name;
        this.birthDate = birthDate;
    }
    //...
}
```
Write a program which reads the data of several persons and constructs an array of Persons.
Relationships between classes

Relationship cardinality

- One-to-one

- One-to-many
Implementing *one-to-many* relationship (1)

```java
public class Student{
    private final long ID;
    private String firstname;
    private String lastname;
    //...
}
```

```java
public class Course{
    private final long ID;
    private String name;
    public static final int MAX_STUDENTS=100;
    private Student[] enrolledStudents;
    private int numStudents;
    //...
}
```
Implementing one-to-many relationship (2)

```java
public class Course{
    private final long ID;
    private String name;
    public static final int MAX_STUDENTS = 100;
    private Student[] enrolledStudents;
    private int numStudents;

    public Course( long ID, String name ){
        this.ID = ID;
        this.name = name;
        enrolledStudents = new Student[ MAX_STUDENTS ];
    }

    public void enrollStudent( Student student ){
        enrolledStudents[ numStudents ] = student;
        ++numStudents;
    }

    //...
}
```
Properties:

- ID
- name

Methods:

- enrollStudent

```java
class Course{
  private final long ID;
  private String name;

  private ArrayList<Student> enrolledStudents;

  public Course( long ID, String name ){
    this.ID = ID;
    this.name = name;
    enrolledStudents = new ArrayList<Student>();
  }

  public void enrollStudent( Student student ){
    enrolledStudents.add(student);
  }

  //...
}
```
Module 5
Inheritance, Polymorphism
Outline

- Inheritance
  - Parent class
  - Subclass, Child class
- Polymorphism
  - Overriding methods
  - Overloading methods
  - The `instanceof` operator
  - Heterogeneous collections
Problem: repetition in implementations

class Employee{
    private String name;
    private double salary;
    private Date birthDate;

    public String toString(){
        //...
    }
}

class Manager{
    private String name;
    private double salary;
    private Date birthDate;
    private String department;

    public String toString(){
        //...
    }
}
public class Employee{
  protected String name;
  protected double salary;
  protected Date birthDate;
  public Employee(...){
    // ...
  }
  public String toString(){
    //...
  }
}

public class Manager extends Employee{
  private String department;

  public Manager(...){
    // ...
  }
  public String toString(){
    // ...
  }
}
Inheritance - syntax

```java
<modifier> class <name> extends <superclass>{
   <declaration*>
}
```

```java
public class Manager extends Employee{
}
```
The subclass

- **Inherits** the **data** and **methods** of the **parent class**
- **Does not inherit the constructors** of the **parent class**
- **Opportunities:**
  1) add new data
  2) add new methods
  3) override inherited methods (polymorphism)
The subclass

- Opportunities:
  1) add new data → department
  2) add new methods → e.g. getDepartment()
  3) override inherited methods → toString()
public class Employee {
    protected String name;
    protected double salary;
    protected Date birthDate;
    public Employee(String name, double salary, Date birthDate) {
        this.name = name;
        this.salary = salary;
        this.birthDate = birthDate;
    }
    //...
}

public class Manager extends Employee {
    private String department;
    public Manager(String name, double salary, Date birthDate, String department) {
        super(name, salary, birthDate);
        this.department = department;
    }
    //...
}
## Access Control

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Same Class</th>
<th>Same Package</th>
<th>Subclass</th>
<th>Universe</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>default</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>protected</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Polymorphism - Overriding Methods

- A subclass can modify the behavior inherited from a parent class.
- A subclass can create a method with different functionality than the parent's method but with the:
  - same name
  - same argument list
  - almost the same return type

(can be a subclass of the overriden return type)
public class Employee{
    protected String name;
    protected double salary;
    protected Date birthDate;
    public Employee( ... ){  
        // ...
    }  
    public String toString(){
        return "Name: " + name + " Salary: " + salary + " B. Date:" + birthDate;
    }
}

class Manager extends Employee{
    private String department;
    public Manager( ... ){  
        // ...
    }  
    @Override
    public String toString(){
        return "Name: " + name + " Salary: " + salary + " B. Date:" + birthDate + " department: " + department;
    }
}
public class Employee{
    protected String name;
    protected double salary;
    protected Date birthDate;
    public Employee( ... ){  
        // ...
    }
    public String toString(){
        return “Name: “+name+” Salary: “+salary+” B. Date:”+birthDate;
    }
}

public class Manager extends Employee{
    private String department;
    public Manager( ... ){  
        // ...
    }
    public String toString(){
        return super.toString() + ” department: “+department;
    }
}
Overridden Methods Cannot Be Less Accessible

```java
public class Parent{
    public void foo()
}

public class Child extends Parent{
    private void foo(){} //illegal
}
```
Overriding Methods

- **Polymorphism**: the ability to have many different forms

```java
Employee e = new Employee(...);
System.out.println( e.toString() );
e = new Manager(...);  //Correct
System.out.println( e.toString() );
```

Which `toString()` is invoked?
public String createMessage(Employee e) {
    return "Hello, " + e.getName();
}

//...
Employee e1 = new Employee("Endre", 2000, new Date(20, 8, 1986));
Manager m1 = new Manager("Johann", 3000,
        new Date(15, 9, 1990), "Sales");

//...
System.out.println(createMessage(e1));
System.out.println(createMessage(m1));

Liskov Substitution!
Employee emps[] = new Employee[100];
emps[0] = new Employee();
emps[1] = new Manager();
emps[2] = new Employee();
// ...

// print employees
for (Employee e: emps) {
    System.out.println(e.toString());
}

// count managers
int counter = 0;
for (Employee e: emps) {
    if (e instanceof Manager) {
        ++counter;
    }
}


Static vs. Dynamic type of a reference

// static (compile time) type is: Employee
Employee e;

// dynamic (run time) type is: Employee
e = new Employee();

// dynamic (run time) type is: Manager
e = new Manager();
Static vs. Dynamic type of a reference

Employee e = new Manager("Johann", 3000,
    new Date(10, 9, 1980), "sales");
System.out.println(e.getDepartment()); // ERROR

// Solution
System.out.println(((Manager) e).getDepartment()); // CORRECT

// Better Solution
if (e instanceof Manager){
    System.out.println(((Manager) e).getDepartment());
}
The `instanceof` Operator

```java
Animal a = new Bear();

//expressions
a instanceof Animal → true
a instanceof Mammal → true
a instanceof Bear → true
a instanceof Date → false
```
Polymorphism

Overloading Methods

- *Polymorphism*: the ability to have many different forms
- Methods overloading:
  - methods having the **same name**,
  - argument list **must** differ,
  - return types **can be** different.
- Example:
  
  ```java
  public void println(int i)
  public void println(float f)
  public void println(String s)
  ```
Polymorphism

Overloading Constructors

```java
public class Employee{
    protected String name;
    protected double salary;
    protected Date birthDate;
    public Employee( String name, double salary, Date birthDate){
        this.name = name;
        this.salary = salary;
        this.birthDate = birthDate;
    }
    public Employee( String name, double salary){
        this(name, salary, null);
    }
    public Employee( String name, Date birthDate){
        this(name, 1000, birthDate);
    }
    //...
}
```
Remember

- Inheritance
  - Subclass opportunities

- Polymorphism
  - *Overriding* methods
  - *Overloading* methods
  - *Polymorphic* argument
  - *Heterogeneous* collections
  - Static vs. dynamic type
  - The `instanceof` operator
Inheritance and Polymorphism

Methods Common to All Objects

- The `equals` method
- The `toString` method
- The `clone` method
Object is a concrete class with (equals, toString, clone, ...) non final methods

- It is designed for extension
- Its methods have explicit general contracts
The equals method

- In class Object equals tests object identity

```java
MyDate s1 = new MyDate(20, 10, 2016);
MyDate s2 = new MyDate(20, 10, 2016);
System.out.println(s1.equals(s2));
s1 = s2;
System.out.println(s1.equals(s2));
```

Output?
public class MyDate {
    private int day;
    private int month;
    private int year;

    public boolean equals(Object o) {
        boolean result = false;
        if ( (o != null) && (o instanceof MyDate) ) {
            MyDate d = (MyDate) o;
            if ((day == d.day) &&
                (month == d.month) &&
                (year == d.year)) {
                result = true;
            }
        }
        return result;
    }
}
The equals method

- In class MyDate equals tests **object logical equality**

```java
MyDate s1 = new MyDate(20, 10, 2016);
MyDate s2 = new MyDate(20, 10, 2016);
System.out.println( s1.equals(s2));
s1 = s2;
System.out.println( s1.equals(s2));
```

Output?
The `equals` method implements an equivalence relation

- **Reflexive**
  - `x.equals(x):true`

- **Symmetric**
  - `x.equals(y):true ↔ y.equals(x):true`

- **Transitive**
  - `x.equals(y):true and y.equals(z):true → x.equals(z):true`
The `toString` method

- Characteristics:
  - Converts an object to a `String`
  - Override this method to provide information about a user-defined object in readable format
# Wrapper Classes

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
</tbody>
</table>
Wrapper Classes

Boxing and Unboxing

```java
int i = 420;
Integer anInt = new Integer(i); // boxing
int j = anInt.intValue(); // unboxing
```
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i < Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
Module 6

Interfaces and Abstract Classes
Outline

- Interfaces
- Interfaces (since Java 8)
- Abstract classes
- Sorting
  - Comparable interface
  - Comparator interface
Interfaces

- Properties
  - Define types
  - Declare a set of methods *no implementation!* – ADT – Abstract Data Type
  - Will be implemented by classes
The Driveable Interface

public interface Driveable{
    public void start();
    public void forward();
    public void turn(double angle);
    public void stop();
}

class interfaces

«interface»
Driveable

+ start() : void
+ forward() : void
+ turn(double) : void
+ stop() : void
public class Bicycle implements Driveable{
    @Override
    public void start() {
        System.out.println("The bicycle has been started");
    }

    @Override
    public void forward() {
        System.out.println("The bicycle moves forward");
    }

    @Override
    public void turn( double angle) {
        System.out.println("The bicycle turns "+angle+" clockwise");
    }

    @Override
    public void stop() {
        System.out.println("The bicycle has been stopped");
    }
}

Implementing the Driveable Interface
Interfaces

- The interface contains **method declarations** and may contain constants
- All the methods are **public** (even if the modifier is missing)
- Interfaces are **pure abstract classes** → cannot be instantiated
- The implementer classes should **implement all the methods** declared in the interface
- A **class can extend a single class** but may implement any number of interfaces
Select the correct statements!

a)  Driveable a;
b)  Driveable a = new Driveable();
c)  Driveable t[] = new Driveable[3];
d)  public void drive( Driveable d );
Interfaces vs. Classes

● **Interface:**
  - User-defined type
  - Set of methods
  - No implementations provided
  - Cannot be instantiated

● **Class:**
  - User-defined type
  - Set of data and methods
  - All the methods are implemented
  - Can be instantiated
public class Utils{

    public void static void moveMe(Driveable v){
        v.start();
        for( int i=0; i<12; ++i){
            v.turn(15);
        }
        v.stop();
    }
}

Utils.moveMe( new Bicycle() );
Utils.moveMe( new Car() );

What am I doing?
public class Utils{
    public static void printIt(List<String> list){
        for( String s: list ){
            System.out.println( s );
        }
    }
}

ArrayList<String> l1 = new ArrayList<>();
// add elements to l1
Utils.printIt(l1);

LinkedList<String> l2 = new LinkedList<>();
// add elements to l2
Utils.printIt(l2);
Interfaces Java 8

- Java Interface **Default** Method
- Java Interface **Static** method
public interface Animal{
    // Abstract method
    void eat();
    // Implemented method
    default void log( String str ){
        System.out.println("Animal log: "+str);
    }
}
public class Bear implements Animal{
   // Mandatory!!!
   void eat(){
      System.out.println("Bear eats");
   }
   // It is not mandatory to provide
   // implementation for the log method
}
public interface MatrixOperations{

    static Matrix add(Matrix a, Matrix b){
        //...
    }

}
public interface MatrixOperations{
    
    static Matrix add(Matrix a, Matrix b) {
        // ...
    }

}
public interface MatrixOperations{

    static Matrix add(Matrix a, Matrix b){
        //...
    }

}

Helper methods – associated with class, not instances

Cannot be overridden in implementer classes
Abstract Classes

- May contain **abstract** and **implemented** methods as well
- May contain **data**
- Cannot be instantiated
- Are designed for subclassing
Abstract Classes

```plaintext
class class8_1

Shape
# area: double
+ draw(): void

Circle
- radius: double
+ Circle(double)
+ draw(): void

Square
- size: double
+ Square(double)
+ draw(): void
```
Abstract Classes

```java
public abstract class Shape {
    protected double area;
    public abstract void draw();
}

public class Square extends Shape {
    private double size;
    public Square(double size) {
        this.size = size;
        this.area = size * size;
    }

    @Override
    public void draw() {
        System.out.println("I am a square");
    }
}
```
Abstract Classes vs. Classes

- **Abstract class:**
  - User-defined type
  - Set of data and methods
  - Abstract and implemented methods
  - **Cannot be instantiated**
  - Designed to be subclassed

- **Class:**
  - User-defined type
  - Set of data and methods
  - All the methods are implemented
  - **Can be instantiated**
## Abstract Classes vs. Classes vs. Interfaces

<table>
<thead>
<tr>
<th></th>
<th>Interface</th>
<th>Abstract class</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract method</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Implemented method</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Yes(since Java 8)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Attribute</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Constants (final)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Sorting and Interfaces

- Sorting Strings, primitives
  - `Arrays.sort()`
  - `Collections.sort()`

- Sort **user-defined** types
  - The `Comparable` interface
  - The `Comparator` interface
Sorting and Interfaces

https://www.mkyong.com/java/java-object-sorting-example-comparable-and-comparator/
Sorting Collections

- Sorting objects by their **natural order**
  - The `Comparable` interface

- Sorting objects using a Comparator
  - The `Comparator` interface
The Comparable interface

interface Comparable {
    int compareTo(Object o);
}

x.compareTo(y):
    0: x equal to y
    positive: x > y;
    negative: x < y;
The Comparable\(\langle T \rangle\) interface

```java
interface Comparable\(\langle T \rangle\) {
    int compareTo(T o);
}
```

Attempts to use a different type are caught at compile time!!!
public class Point implements Comparable<Point> {
    //...
    @Override
    public int compareTo(Point o) {
        if (o == null ) throw new NullPointerException();
        if (this.x == o.x && this.y == o.y) {
            return 0;
        }
        if( this.x == o.x){
            return this.y - o.y;
        }
        return this.x - o.x;
    }
}
The Comparable\(\langle T \rangle\) interface

Consistency

If a class overrides the `equals` method, then it is advisable (but not enforced) that

\[
a.\text{equals}(b)
\]

exactly when

\[
a.\text{compareTo}(b) == 0
\]
What if we need multiple sorting criteria?

- Class **Point**
  - Sorting by $x$ then by $y$
  - Sorting by $y$ then by $x$
  - Sorting by the distance from the origin $(0,0)$

- For each class we can define **only one natural ordering** through the **Comparable** interface

- We can define an **unlimited number of ordering** using the **Comparator** interface
The Comparator\langle T \rangle interface

interface Comparator\langle T \rangle {
    int compare (T x, T y);
}

class DistanceComparator implements Comparator<Point>
{
    private final Point origo = new Point(0,0);

    @Override
    public int compare(Point p1, Point p2) {
        Double d1 = p1.distanceTo(origo);
        Double d2 = p2.distanceTo(origo);
        return d1.compareTo(d2);
    }
}

ArrayList<Point> points = new ArrayList<Point>();
points.add( new Point(1,2));
points.add( new Point(2,2));
points.add( new Point(1,3));

Collections.sort( points, new DistanceComparator() );
for( Point point: points ){
    System.out.println(point);
}
The Comparator<T> interface (2)
Anonymous inner class

```java
ArrayList<Point> points = new ArrayList<>();
points.add(new Point(1, 2));
points.add(new Point(2, 2));
points.add(new Point(1, 3));
Collections.sort(points, new Comparator<Point>() {
    private final Point origo = new Point(0, 0);
    @Override
    public int compare(Point p1, Point p2) {
        Double d1 = p1.distanceTo(origo);
        Double d2 = p2.distanceTo(origo);
        return d1.compareTo(d2);
    }
});
for (Point point : points) {
    System.out.println(point);
}
```
The Comparator<> interface (3)
Lambda

```java
ArrayList<Point> points = new ArrayList<>();
points.add(new Point(1, 2));
points.add(new Point(2, 2));
points.add(new Point(1, 3));

Collections.sort(points,
        (Point p1, Point p2) ->
        ((Double) p1.distanceTo(new Point(0, 0))).compareTo((Double) p2.distanceTo(new Point(0, 0)));

for (Point point : points) {
    System.out.println(point);
}
```
Module 7
Exceptions
Exceptions

- Define exceptions
- Exception handling: try, catch, and finally
- Exception categories
- User-defined exceptions
- Enumerations
- Nested classes
public class AddArguments {
    public static void main(String[] args) {
        int sum = 0;
        for (String arg: args) {
            sum += Integer.parseInt(arg);
        }
        System.out.println( "Sum: " + sum );
    }
}

java AddArguments 1 2 3
Sum: 6

java AddArguments 1 foo 2 3

Exception in thread "main" java.lang.NumberFormatException: For input string: "foo"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Integer.parseInt(Integer.java:580)
at java.lang.Integer.parseInt(Integer.java:615)
at addarguments.AddArguments.main(AddArguments.java:line_number)
Java Result: 1
The try-catch statement

```java
public class AddArguments2 {
    public static void main(String[] args) {
        try {
            int sum = 0;
            for (String arg: args) {
                sum += Integer.parseInt(arg);
            }
            System.out.println("Sum: "+sum);
        } catch (NumberFormatException e) {
            System.err.println("Non-numeric argument");
        }
    }
}
```

```
java AddArguments2 1 foo 2 3
Non-numeric argument
```
The try-catch statement

```java
public class AddArguments3 {
    public static void main(String[] args) {
        int sum = 0;
        for (String arg : args) {
            try {
                sum += Integer.parseInt(arg);
            } catch (NumberFormatException e) {
                System.err.println(arg + " is not an integer");
            }
        }
        System.out.println("Sum: "+sum);
    }
}
```
```
java AddArguments3   1 foo 2 3  
foo is not an integer
Sum: 6
```
The *try-catch* statement

```java
try{
    // critical code block
    // code that might throw exceptions
} catch( MyException1 e1 ){
    // code to execute if a MyException1 is thrown
} catch( MyException2 e2 ){
    // code to execute if a MyException1 is thrown
} catch ( Exception e3 ){
    // code to execute if any other exception is thrown
}
```
Call Stack Mechanism

- If an exception is not handled in a method, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.
try{
    connectDB();
    doTheWork();
} catch( AnyException e ){
    logProblem( e );
} finally {
    disconnectDB();
}

The code in the **finally** block is always executed (even in case of return statement)
Exception Categories

- Checked and unchecked exceptions

![Exception Categories Diagram]
The Handle or Declare Rule

```java
public static int countLines( String filename ){
    int counter = 0;
    Scanner scanner = null;
    try {
        scanner = new Scanner( new File( filename ));
    } catch (FileNotFoundException ex) {
        ex.printStackTrace();
    }
    String line;
    while ( scanner.hasNextLine() ){
        line = scanner.nextLine();
        ++counter;
    }
    return counter;
}
```

Usage:
```java
ClassName.countLines("input.txt");
```
public static int countLines( String filename )
throws FileNotFoundException{
    int counter = 0;
    Scanner scanner = new Scanner(new File( filename ));
    String line;
    while ( scanner.hasNextLine() ){
        line = scanner.nextLine();
        ++counter;
    }
    return counter;
}

Usage:
try{
    ClassName.countLines("input.txt");
} catch( FileNotFoundException e ){
    e.printStackTrace();
}
void trouble1 () throws Exception1 {...}
void trouble2 () throws Exception1, Exception2 {...}

Principles:

- You do not need to declare runtime (unchecked) exceptions
- You can choose to handle runtime exceptions (e.g. IndexOutOfBoundsException, NullPointerException)
Creating Your Own Exceptions

The overriding method can throw:
- No exceptions
- One or more of the exceptions thrown by the overridden method
- One or more subclasses of the exceptions thrown by the overridden method

The overridden method cannot throw:
- Additional exceptions not thrown by the overridden method
- Superclasses of the exceptions thrown by the overridden method
public class StackException extends Exception {
    public StackException(String message) {
        super(message);
    }
}

User-Defined Exception
public class Stack {
    private Object elements[];
    private int capacity;
    private int size;

    public Stack(int capacity) {
        this.capacity = capacity;
        elements = new Object[capacity];
    }

    public void push(Object o) throws StackException {
        if (size == capacity) {
            throw new StackException("Stack is full");
        }
        elements[size++] = o;
    }

    public Object top() throws StackException {
        if (size == 0) {
            throw new StackException("stack is empty");
        }
        return elements[size - 1];
    }
    // ...
}
Stack s = new Stack(3);
for (int i = 0; i < 10; ++i) {
    try {
        s.push(i);
    } catch (StackException ex) {
        ex.printStackTrace();
    }
}
Module 8

Nested Classes
Nested Classes

- **When?**
  - If a class is used only inside of another class (encapsulation)
  - Helper classes
Nested Classes

- **The place of nesting**
  - Class
  - Method
  - Instruction

- **Embedding method**
  - Static
  - Non-static
public class Slist{
    private Element head;

    public void insertFirst( Object value ){
        head = new Element(value, head);
    }

    private static class Element{
        private Object value;
        private Element next;
        public Element( Object value, Element next){
            this.value = value;
            this.next = next;
        }
        public Element( Object value){
            this.value = value;
            this.next = null;
        }
    }
}
The Iterator interface

Package: java.util

public interface Iterator{
    public boolean hasNext();
    public Object next();
    //optional
    public void remove();
}

Make Slist iterable using the Iterator interface!!!
The Iterator interface

Slist list = new Slist();
for( int i=0; i<10; ++i ){
    list.insertFirst( i );
}

Iterator it = list.createIterator();
while( it.hasNext() ){
    System.out.println( it.next() );
}
public class Slist{
    private Element head;
    //...

    public Iterator createIterator(){
        return new ListIterator();
    }

    private class ListIterator implements Iterator{
        private Element act = head;
        public boolean hasNext(){
            return act != null;
        }
        public Object next(){
            Object value = act.value;
            act = act.next;
            return value;
        }
    }
}
public class Slist{
    private Element head;
    //...

    public Iterator createIterator(){
        return new ListIterator();
    }

    private class ListIterator implements Iterator{
        private Element act = head;
        public boolean hasNext(){
            return act != null;
        }
        public Object next(){
            Object value = act.value;
            act = act.next;
            return value;
        }
    }
}
public class Slist{
    private Element head;
    //...

    public Iterator createIterator(){
        return new Iterator(){
            private Element act = head;

            public boolean hasNext(){
                return act != null;
            }

            public Object next(){
                Object value = act.value;
                act = act.next;
                return value;
            }
        }
    }
}
Module 9
Threads
Outline

- Definition
- Creation: *Thread* and *Runnable*
- Synchronization
- Executors and thread pools
What are threads?

- Operating Systems
  - lightweight process
  - runs in the address space of a process
  - has its own program counter (PC)+stack
  - shares code and data with other threads

- Object-oriented Programming
  - an object – an instance of the class Thread
Threads

java.lang.Thread = Infrastructure(PC+Stack)

java.langRunnable = Code
public class MyRunnable implements Runnable{
    private int id;

    public MyRunnable(int id ){
        this.id = id;
    }

    public void run(){
        for( int i=0; i<10; ++i){
            System.out.println("Hello"+id+" "+i);
        }
    }
}

... 
MyRunnable r = new MyRunnable(1); 
Thread t = new Thread( r );
Starting the thread

Thread t = new Thread(r);
Constructor initializes the thread object

t.start();
Calls the thread object's run method
public class Test{
    public static void main(String args[]){
        Thread t1 = new Thread( new MyRunnable(1));
        Thread t2 = new Thread( new MyRunnable(2));
        t1.start();
        t2.start();
    }
}

Output?
```java
class MyThread extends Thread {
    private int id;

    public MyThread(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; ++i) {
            System.out.println("Hello" + id + " " + i);
        }
    }
}

Thread t = new MyThread(1);
t.start();
```
public class Test {
    public static void main(String[] args) {
        Thread t1 = new MyThread(1);
        Thread t2 = new MyThread(2);
        t1.start();
        t2.start();
    }
}
public class **MyFirstRunnable** implements **Runnable**{
    @Override
    public void run() {
        System.out.println("In a thread");
    }
}

**Usage:**

Thread thread = new Thread(new MyFirstRunnable());
thread.start();
System.out.println("In the main Thread");

**Output?**
public class MyFirstRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("In a thread");
    }
}

Usage:
Runnable runnable = new MyFirstRunnable();
for(int i = 0; i<25; i++){
    new Thread(runnable).start();
}

How many threads?
public class **MyFirstRunnable** implements **Runnable**{
    @Override
    public void run() {
        System.out.println("In a thread");
    }
}

**Usage:**

Thread thread = new Thread(new MyFirstRunnable());
thread.run();
System.out.println("In the main Thread");
Operations on threads

- make the current Thread **sleep**
- wait for another thread to complete (**join**)
- manage the **priorities** of threads
- **interrupt** a thread
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
sleep()

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- It always pause the current thread execution.
- The actual time thread sleeps depends on system timers and schedulers (for a busy system, the actual time for sleep is a little bit more than the specified sleep time).
Thread \texttt{t2} = new Thread(new R());
\texttt{t2.start();}
try {
    \texttt{t2.join();}
} catch (InterruptedException \texttt{e}){
    \texttt{e.printStackTrace();}
}
public class ThreadPriorityRange {
    public static void main(String[] args) {
        System.out.println("Minimal priority : " + Thread.MIN_PRIORITY);
        System.out.println("Maximal priority : " + Thread.MAX_PRIORITY);
        System.out.println("Norm priority : " + Thread.NORM_PRIORITY);
    }
}
A thread can be interrupted:

- if the thread is sleeping
- if the thread is waiting for another thread to join
private static class **ForeverRunnable** implements Runnable {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName() + 
                "\t": " + System.currentTimeMillis());
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                System.out.println(
                    Thread.currentThread().getName() + 
                    "\thas been interrupted");
            }
        }
    }
}
private static class **ForeverRunnable** implements Runnable {
    public void run() {
        while (true) {
            System.out.println(Thread.currentThread().getName() + 
            " : " + System.currentTimeMillis());
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + 
                "has been interrupted");
            }
        }
    }
}

public static void main(String[] args) {
    Thread t2 = new Thread(new **ForeverRunnable**());
    System.out.println("Current time millis : " + 
    System.currentTimeMillis());
    t2.start();
    t2.*interrupt*();
}
Thread's states

- New
  - start()
  - Runnable:
    - sleep, block on I/O, wait for lock, suspend, wait
    - sleep done, I/O complete, lock available, resume, notify
  - Runnable
  - Non-Runnable (Blocked)
  - Running
  - Terminated

run() method exits
Need for synchronization

public class Counter {
    private int value = 0;

    public int getNextValue() {
        return value++;
    }
}

Thread1
public class Counter {
    private int value = 0;

    public int getNextValue() {
        return value++;
    }
}

Thread1

Thread2
class Counter {
    private int value;

    public int getNextValue() {
        return ++value;
    }
    public int getValue(){
        return value;
    }
}
class Thread3 extends Thread {
    private Counter counter;

    public Thread3(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 0; i < 10000; ++i){
            counter.getNextValue();
        }
    }
}
Counter counter = new Counter();
Thread t1 = new Thread3(counter);
Thread t2 = new Thread3(counter);
t1.start();
t2.start();
try{
    t1.join();
    t2.join();
} catch( InterruptedException e ){
}
System.out.println("COUNTER: "+counter.getValue());
Need for synchronization

value++

1. Read the current value of "value"
2. Add one to the current value
3. Write that new value to "value"
public class Counter {
    private int value = 0;

    public synchronized int getNextValue() {
        return value++;
    }
}
public class Counter {
    private int value = 0;

    public int getNextValue() {
        synchronized (this) {
            value++;
        }
        return value;
    }
}
Synchronized Blocks

- every object contains a single lock
- the lock is taken when synchronized section is entered
- if the lock is not available, thread enters a waiting queue
- if the lock is returned, thread is resumed
A class is **thread safe** if it behaves always in the same manner when accessed from multiple threads.

Stateless objects (immutable classes) are always thread safe:
- String
- Long
- Double
Executors and thread pools

http://www.vogella.com/tutorials/JavaConcurrency/article.html#threadpools

ExecutorService
Callable
Future
Module 10
GUI Programming
Swing
Java GUIs

- **AWT (Abstract Windowing Toolkit)** – since JDK 1.0
  - Uses native control
  - Appearance/behavior depends on platform
- **Swing** – since JDK 1.2
  - Implemented completely in Java (light weight)
- **JavaFX** – since JDK 8
  - Written as a native library
  - Provided on a wide variety of devices
- **SWT (Standard Widget Toolkit)**
  - Eclipse
Outline

- Containers, components and layout managers
- FlowLayout, BorderLayout, and GridLayout
- Add components to a container
- Events and event handling
- Delegation model
- Adapter classes
Component

- Represents an object with visual representation
- Other names for components: widgets, controls

![Diagram](image)
Container

- A special component that holds other components
- Used for grouping other components
public static void main(String[] args) {
    JFrame f = new JFrame("The First Swing Application");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setBounds(100, 100, 300, 300);
    f.setVisible(true);
}
Frames

JFrame

- Top level container
  - can have menu bars
- Contains a JRootPane
- Have title and resizing corners
- Have BorderLayout as the default layout manager
Positioning Components

- Responsibility of the layout manager
  - size (dimension: width and height in pixels)
  - position (location of the top left corner)

- You can disable the layout manager:
  `setLayout(null),`

  then use

  - `setSize() + setLocation()`
  - `setBounds()`
Organizing Components (1)

```java
JFrame f = new JFrame("The First Swing Application");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel p = new JPanel();
p.setBackground(Color.blue);
JButton b = new JButton("Yes");
p.add(b);
f.setContentPane(p);

f.setBounds(100, 100, 300, 300);
f.setVisible(true);
```
```java
JFrame f = new JFrame("The First Swing Application");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel p = new JPanel();
p.setBackground(Color.blue);
p.setLayout(null);
JButton b = new JButton("Yes");
b.setSize(100, 60);
b.setLocation(200, 200);
p.add(b);
f.setContentPane(p);
f.setBounds(100, 100, 300, 300);
f.setVisible(true);
```
Layout Managers

- FlowLayout
- BorderLayout
- GridLayout
- GridBagLayout
Layout Managers

GridLayout

public static JPanel createPanel( int n) {
    JPanel panel = new JPanel();
    panel.setLayout(new GridLayout( n, n));
    for( int i=0; i<n; ++i) {
        for( int j=0; j<n; ++j) {
            panel.add( new JButton
                ("("+i+","+j+")");
        }
    }
    return panel;
}
Creating UI

- Aggregation
  - FrameAggregation
- Inheritance
  - FrameInheritance
public class FrameAggregation {

    private static void initFrame() {
        JFrame frame = new JFrame("FrameAggregation");
        frame.add(new JButton("Ok"), "Center");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setBounds(100, 100, 200, 200);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        initFrame();
    }
}
public class FrameInheritance extends JFrame {
    private JButton button;
    public FrameInheritance() {
        initComponents();
    }
    private void initComponents() {
        this.setTitle("FrameInheritance");
        this.add(new JButton("Ok"), "Center");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setBounds(100, 100, 200, 200);
        this.setVisible(true);
    }
    public static void main(String[] args) {
        new FrameInheritance();
    }
}
private static JMenuBar createMenu() {
    //MenuBar
    MenuBar menuBar; JMenu filemenu, helpmenu;
    JMenuItem menuItem;
    menuBar = new JMenuBar();
    // Build File menu.
    filemenu = new JMenu("File"); menuBar.add(filemenu);
    menuItem = new JMenuItem("New"); filemenu.add(menuItem);
    menuItem = new JMenuItem("Exit"); filemenu.add(menuItem);
    // Build Help menu.
    helpmenu = new JMenu("Help");
    menuBar.add(helpmenu);
    menuItem = new JMenuItem("About");
    helpmenu.add(menuItem);
    return menuBar;
}

frame.setJMenuBar(createMenu());
Dialogs

**JOptionPane (1)**

```java
JOptionPane.showMessageDialog(
    Component parent, String message);
```
Dialogs

JOptionPane (2)

```java
int result = JOptionPane.showConfirmDialog(
    Component parent, String message);
Result:
    YES_OPTION (0), NO_OPTION (1), CANCEL_OPTION (2)
```
Dialogs

JOptionPane (3)

String value=
    JOptionPane.showMessageDialog("Please input a value");
String options[]={"Apple", "Grape", "Strawberry"};

int res = JOptionPane.showMessageDialog(form, "Choose an option", "OptionDialog", JOptionPane.DEFAULT_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[0]);
Dialogs
Chooser

JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(parent);
if (returnVal == JFileChooser.APPROVE_OPTION) {
    System.out.println(
        "You chose to open this file: " +
        chooser.getSelectedFile().getName());
}
Borders

JPanel pane = new JPanel();
pane.setBorder(BorderFactory.createLineBorder(Color.black));

http://docs.oracle.com/javase/tutorial/uiswing/components/border.htm
Custom properties

- (key, value) pairs associated to JComponent type objects
  - Key: Object
  - Value: Object

```java
JButton button = new JButton("Press Me");
button.putClientProperty("order","10");
//...
button.getClientProperty("order");
```
Event Handling

- **Event** — objects that describe what happened
- **Event source** — the generator of an event
- **Event handler** — a method that
  - receives an event object,
  - deciphers it,
  - and processes the user's interaction

```java
Frame
Panel

The user clicks on the button

Evento

EventoPerformed(ActionEvent e){
   ...
}
```
Event Types

- Low level
  - Window
  - Keyboard
  - Mouse
- High level
  - ActionEvent
  - ItemEvent
- **One event – many handlers**

- Event handlers are registered by event source components

```java
Frame

Panel

Button

ActionEvent

actionPerformed(ActionEvent e){
    ...
}

Event Handler1

Event Handler2

actionPerformed(ActionEvent e){
    ...
}

The user clicks on the button

Frame

Panel

Button

ActionEvent

actionPerformed(ActionEvent e){
    ...
}

Event Handler1

Event Handler2

actionPerformed(ActionEvent e){
    ...
}

The user clicks on the button
Delegation Model

- Client objects (handlers) register with a GUI component that they want to observe.
- GUI components trigger the handlers for the type of event that has occurred.
- Components can trigger more than one type of events.
Delegation Model

JButton b = new JButton("Yes");
f.add(b);
b.addActionListener(new ActionListener(){
    @Override
    public void actionPerformed(ActionEvent e) {
        if( b.getText().equals("Yes") ){
            b.setText("No");
        }else{
            b.setText("Yes");
        }
    }
});

(I) Definition of an anonymous inner class which implements ActionListener interface
(II) Creation of an instance from that anonymous inner class
(III) This instance is responsible for event handling
Delegation Model

Java 8 - Lambdas

```java
JButton b = new JButton("Yes");
f.add( b );
b.addActionListener(e->
{
    b.setText( b.getText().equals("No") ? "Yes": "No");
}
);
```

ActionEvent
public class MyFrame implements ActionListener{
	// ...
	public void initComponents(){
		for( int i=0; i<n; ++i){
			for( int j=0; j<n; ++j){
				JButton b = new JButton(""));
				panel.add( b);
				b.addActionListener( this );
			}
	}
	}

	@Override
	public void actionPerformed(ActionEvent e) {
		JButton source = (JButton) e.getSource();
		source.setBackground(Color.red);
	}
}
public class DrawComponent extends JComponent{
    private ArrayList<Point> points= new ArrayList<Point>();
    private Color color = Color.red;

    public DrawComponent(){
        this.addMouseListener(new MouseAdapter(){
            @Override
            public void mousePressed(MouseEvent e) {
                points.clear();
                points.add( new Point( e.getX(), e.getY()));
            }
        });
        this.addMouseMotionListener(new MouseMotionAdapter(){
            @Override
            public void mouseDragged(MouseEvent e) {
                points.add( new Point( e.getX(), e.getY()));
                DrawComponent.this.repaint();
            }
        });
    }
    ...
}
public class DrawComponent extends JComponent{
    //...
    @Override
    public void paint(Graphics g) {
        g.setColor(color);
        if( points != null && points.size()>0){
            Point startPoint = points.get(0);
            for( int i=1; i<points.size(); ++i ){
                Point endPoint = points.get(i);
                g.drawLine(startPoint.x, startPoint.y,
                            endPoint.x, endPoint.y);
                startPoint = endPoint;
            }
        }
    }
    public void clear(){
        points.clear();
        repaint();
    }
}
Event listeners

- **General listeners**
  - ComponentListener
  - FocusListener
  - MouseListener

- **Special listeners**
  - WindowListener
  - ActionListener
  - ItemListener
Event adapter classes

- **Problem:**
  - Sometimes you need only one event handler method, but the listener interface contains several ones
  - You have to implement all methods, most of them with empty ones

- **Solution:**
  - An Event Adapter is a convenience class
  - Implements all methods of a listener interface with empty methods
  - You extend the adapter class and override that specific method
public class MyClass extends JFrame {
    ...
    someObject.addMouseListener(
        new MouseAdapter()
        {
            public void mouseClicked(MouseEvent e) {
                //Event listener implementation
            }
        }
    );
}
Module 11
Collections and Generics
Outline

● Data Structures
● Interfaces: Collection, List, Set, Map, ...
● Implementations: ArrayList, HashSet, TreeMap, ...
● Traversing collections
● Overriding equals and hashCode
● Sorting
● Problems
The Collections API

- **What is?**
  - Unified architecture
    - Interfaces – implementation-independence
    - Implementations – resusable data structures
    - Algorithms – reusable functionality
  - **Best-known examples**
    - C++ Standard Template Library (STL)
    - Smalltalk collections
The Collections API

● Benefits:
  - Reduces programming effort
  - Increases performance
    ● High performance implementations of data structures
  - Fosters software reuse
The Collections API

Design Goals

- Small and simple
- Powerful
- Easily extensible
- Compatible with preexisting collections
- Easy to use
The Collection interface

Methods:
- add(T what): boolean
- remove(T what): boolean
- size(): int
- contains(T what): boolean
- containsAll(Collection c): boolean
- equals(T what): boolean
- iterator(): Iterator
## Implementations

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Hash Table</th>
<th>Resizable Array</th>
<th>Balanced Tree</th>
<th>Linked List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>HashSet</td>
<td></td>
<td>TreeSet</td>
<td>Linked List</td>
</tr>
<tr>
<td>List</td>
<td></td>
<td>ArrayList</td>
<td></td>
<td>Linked List</td>
</tr>
<tr>
<td>Map</td>
<td>HashMap</td>
<td></td>
<td>TreeMap</td>
<td></td>
</tr>
</tbody>
</table>
List implementations

ArrayList

size

capacity

LinkedList

Source
Set implementations

TreeSet

HashSet
Ordered vs. sorted collections

- **Ordered**
  - You can iterate through the collection in a specific (not random) order.
  - Each element has a previous and a next element (except the first and the last ones).

- **Sorted**
  - The order is determined according to some rule or rules (*sort order*).
  - Is a specific type of ordering

- **Collections**
  - `HashSet`: unordered and unsorted
  - `List`: ordered but unsorted
  - `TreeSet`: ordered and sorted
## Complexities

<table>
<thead>
<tr>
<th></th>
<th>add (append)</th>
<th>get (position)</th>
<th>remove</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ArrayList</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>LinkedList</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>HashSet</strong></td>
<td>O(1)*</td>
<td>-</td>
<td>O(1)*</td>
<td>O(1)*</td>
</tr>
<tr>
<td><strong>TreeSet</strong></td>
<td>O(log n)</td>
<td>-</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

* in the case of a proper hash function
Traversing Collections

- There are 3 ways:
  1) for-each
  2) Iterator
  3) Using aggregate operations (*since Java 8*)
Traversing Collections

(1) for-each

```java
ArrayList list1 = new ArrayList();
...
for(Object o: list1){
    System.out.println(o);
}
--------------------------------------------
ArrayList<Person> list2 = new ArrayList<>();
...
for(Person p: list2){
    System.out.println(p);
}
```
package java.util;

public interface Iterator{
    boolean hasNext();
    Object next();
    void remove(); //optional
}

public interface Iterator<E>{
    boolean hasNext();
    E next();
    void remove(); //optional
}
Traversing Collections

(2) Iterator

```java
ArrayList list1 = new ArrayList();
...
Iterator it1 = list1.iterator();
while(it1.hasNext()){
    System.out.println(it1.next());
}

--------------------------------------------

ArrayList<Person> list2 = new ArrayList<>();
...
Iterator<Person> it2 = list2.iterator();
while(it2.hasNext()){
    System.out.println(it2.next());
}
```
Traversing Collections

(2) Iterator

```java
ArrayList list1 = new ArrayList();
...
Iterator it1 = list1.iterator();
while (it1.hasNext()) {
    System.out.println(it1.next());
}

An Iterator is an object
- State: represents a position in a collection
- Behavior: permits to step through the collection

ArrayList<Person> list2 = new ArrayList<>();
...
Iterator<Person> it2 = list2.iterator();
while (it2.hasNext()) {
    System.out.println(it2.next());
}
```
Traversing Collections

(3) Using aggregate operations

Java 8

```java
TreeSet<String> dict = new TreeSet<>();
Scanner scanner = new Scanner( new File("dict.txt"));
while( scanner.hasNext()){
    dict.add( scanner.next());
}
System.out.println("SIZE: "+dict.size());
long counter = dict.stream()
    .filter( e ->
        e.startsWith("the")
    ).count();
System.out.println("#words: "+counter);
```
Problems

Which data structure to use?

Problem:
Split a text file into words and print the words in
1) Increasing order (alphabetically)
2) Decreasing order
Problems

Which data structure to use?

Problem:
Split a text file into words and print the distinct words in
1) Increasing order (alphabetically)
2) Decreasing order

Solutions:
1) TreeSet<String>
2) TreeSet<String> (Comparator<String>)
Problems

Decreasing Order

```java
TreeSet<String> set = new TreeSet<>();
//...
TreeSet<String> rev = new TreeSet<>(
    new Comparator<String>(){
        @Override
        public int compare(String o1, String o2) {
            return o2.compareTo(o1);
        }
    }
);
rev.addAll( set );
```
Problem

Which data structure to use?

Problem:
Generate 2D Points having integer coordinates and print them in increasing order. Points are ordered according to their distance to the origin.
public class Point implements Comparable<Point> {
    public static final Point origin = new Point(0, 0);

    private final int x, y;
    // constructor + getters
    public String toString() { //...}
    public boolean equals(Object obj) { //...}
    public double distanceTo(Point point) { //...}

    @Override
    public int compareTo(Point o) {
        double d = this.distanceTo(origin) - o.distanceTo(origin);
        if (d < 0) return -1;
        else if (d > 0) return 1;
        else return 0;
    }
}
public class Point implements Comparable<Point>{
    public static final Point origin = new Point(0,0);

    private final int x, y;
    // constructor + getters
    public String toString(){ //...}
    public boolean equals(Object obj){ //...}
    public double distanceTo( Point point ){ //...}

    @Override
    public int compareTo(Point o) {
        double d = this.distanceTo(origin) - o.distanceTo(origin);
        if( d < 0 ) return -1;
        else
            if( d>0 ) return 1;
            else return 0;
    }
}

TreeSet<Point> points1 = new TreeSet<>();
// OR
ArrayList<Point> points2 = new ArrayList<>();
Collections.sort(points2);

public double distanceTo( Point point ){ //...}
Problem

Generate randomly $N = 1,000,000$ (one million) distinct bidimensional points $(x, y)$ having positive integer coordinates ($0 \leq x \leq M$, $0 \leq y \leq M$, $M = 1,000,000$).

**Requirements:**
- Optimal solution is required.
- Print the number of duplicates generated.

*Which collection to use?*

*Hint:* Finding an existing element must be fast.
Problem
1. solution - TreeSet

public class Point implements Comparable<Point> {
  ...
  @Override
  public int compareTo(Point o) {
    if (o == null) throw new NullPointerException();
    if (this.x == o.x &&
        this.y == o.y){
      return 0;
    }
    if (this.x == o.x){
      return this.y - o.y;
    } else{
      return this.x - o.x;
    }
  }
}
public class RandomPoints {
    private TreeSet<Point> points =
        new TreeSet<Point>();
    private int duplicates = 0;

    public RandomPoints( int size, int interval) {
        int counter = 0;
        Random rand = new Random(0);
        while( counter < size ){
            int x = Math.abs(rand.nextInt() % interval);
            int y = Math.abs(rand.nextInt() % interval);
            Point p = new Point(x, y);
            if ( points.contains( p )) {
                ++duplicates;
                continue;
            }
            ++counter;
            points.add(p);
        }
    }
    ...
}
Problem

1. solution - TreeSet

public class RandomPoints {
    private TreeSet<Point> points =
        new TreeSet<Point>();
    private int duplicates = 0;

    public RandomPoints( int size,
        int interval)
    {
        int counter = 0;
        Random rand = new Random(0);
        while( counter < size ){
            int x =
                Math.abs(rand.nextInt() % interval);
            int y = Math.abs(rand.nextInt() %
                interval);
            Point p = new Point(x,y);
            if( points.contains( p )){
                ++duplicates;
                continue;
            }
            ++counter;
            points.add(p);
        }
    }
    ...
}

Implementation

Random number generator: seed = 0
N = 1.000.000
M = 10.000
Duplicates: 4976
Time: approx. 3s
Problem
2. solution - HashSet

@Override
public int hashCode() {
    int hash = (x * 31) ^ y;
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Point other = (Point) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return true;
}
Problem
2. solution - HashSet

@Override
public int hashCode() {
    int hash = (x * 31)^ y;
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Point other = (Point) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return true;
}

What happens if we don't override equals?
How many duplicates?

HashSet

- Finding an element: O(1)

Implementation
Random number generator: seed = 0
N = 1.000.000
M = 10.000
Duplicates: 4976
Time: approx. 1s
@Override
public int hashCode() {
    int hash = 1;
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Point other = (Point) obj;
    if (this.x != other.x) {
        return false;
    }
    if (this.y != other.y) {
        return false;
    }
    return true;
}
Problem
2. solution - HashSet

The `hashCode()` contract:

- each time invoked on the same object must return the same value (consistent, can't be random)
- if `x.equals(y) == true`, then `x.hashCode() == y.hashCode()` must be true
- It is legal to have the same hashcode for two distinct objects (collision)
Problem
3. solution

Which collection to use if $M = 2000$

**Hint:** Which is the fastest access time of an element in a collection?
Problem
3. solution

Which collection to use if $M = 2000$

**Hint:** Which is the fastest access time of an element in a collection?

```java
private boolean exists[][] = new boolean[M][M];

public RandomPoints(int size, int interval){
    int counter = 0;
    Random rand = new Random(0);
    while( counter < size ){
        int x = Math.abs(rand.nextInt() % interval);
        int y = Math.abs(rand.nextInt() % interval);
        Point p = new Point(x,y);
        if( exists[x][y]){
            ++duplicates;
            continue;
        }
        ++counter;
        exists[x][y] = true;
    }
}
```
Which collection to use if M = 2000

**Hint:** Which is the fastest access time of an element in a collection?

```java
private boolean exists[][] = new boolean[M][M];

public RandomPoints( int size, int interval){
    int counter = 0;
    Random rand = new Random(0);
    while( counter < size ){
        int x = Math.abs(rand.nextInt() % interval);
        int y = Math.abs(rand.nextInt() % interval);
        Point p = new Point(x,y);
        if( exists[ x ][y ]){
            ++duplicates;
            continue;
        }
        ++counter;
        exists[ x ][ y ] = true;
    }
}
```

**Bidimensional array of booleans**

- Finding an element: O(1)

**Implementation**

- Random number generator: seed = 0
- N = 1.000.000
- M = 2000
- Duplicates: 150002
- Time: **approx. 0.2 s**
interface `Map<K, V>`

- `K` – Key type
- `V` – Value type

Maps keys to values.

Examples:

- **Key**: country, **Value**: capital city
  - Slovenia → Ljubljana
  - Austria → Vienna
  - Hungary → Budapest
  - Romania → Bucharest
Map

Implementations

**HashMap**: unordered, no duplicates

**TreeMap**: ordered by key, no duplicates

<table>
<thead>
<tr>
<th></th>
<th>get</th>
<th>put</th>
<th>remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeMap</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>HashMap</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
<td>$O(1)^*$</td>
</tr>
</tbody>
</table>

* in the case of a proper hash function
Problem

Which data structure to use?

Problem:
Compute the word frequencies in a text. Print the words and their frequencies:

1) alphabetically,

2) in decreasing frequency order.
Problem

Solution (1) alphabetically

class MyLong {
    private long value;
    public MyLong(int value) { this.value = value;}
    public long getValue() { return value;}
    public void setValue(long value) { this.value = value;}
    public void increment() { ++value;}
}

//...
TreeMap<String, MyLong> frequency = new TreeMap<>();
Problem

Solution (2) decreasing frequency order

class **Pair** {
    private String word;
    private long fr;
    // constructor + get and set methods
}

ArrayList<Pair> list = new ArrayList<Pair>();
for (String key : frequency.keySet()) {
    long value = frequency.get(key).getValue();
    list.add(new Pair(key, value));
}
Collections.sort(list, new Comparator<Pair>() {
    @Override
    public int compare(Pair o1, Pair o2) {
        return (int) (o2.getFr() - o1.getFr());
    }
});
Problem

Which data structure to use?

Problem: Find the anagrams in a text file!
Problem

Which data structure to use?

Problem:
Find the anagrams in a text file!

Solution:

- Split the text into words
- Alphabetize the word
  - sent → enst
  - nest → enst
  - tens → enst

- Map<String, List<String>> vs. Map<String, Set<String>>
  - Key: alphabetized word → String
  - Value: words → List<String> or Set<String>
Map<String, Set<String>> groups = new HashMap<>();
// ...

String word = cleanWord(word);
String key = alphabetize(word);
// Find the key
Set<String> group = groups.get(key);
if (group == null) {
    Set<String> newGroup = new HashSet<String>();
    newGroup.add(word);
    group.add(word);
} else {
    group.add(word);
}
**Problem**

**Anagrams**

```java
Map<String, Set<String>> groups = new HashMap<>();

private void printGroups(int size) {
    for (String key : groups.keySet()) {
        Collection<String> group = groups.get(key);
        if (group.size() == size) {
            System.out.print("Key: " + key + " --> ");
            for (String word : group) {
                System.out.print(word + " ");
            }
            System.out.println();
        }
    }
}
```