Java for the Internet

Objectives

Java for the Internet

Objectives

- A little on concepts in object oriented programming

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program
- Simple objects: ”Strings”, Dates

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program
- Simple objects: ”Strings”, Dates
- Streams, reading and writing to/from files, etc

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program
- Simple objects: ”Strings”, Dates
- Streams, reading and writing to/from files, etc
- Exceptions

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program
- Simple objects: ”Strings”, Dates
- Streams, reading and writing to/from files, etc
- Exceptions
- Some typical Java beginner errors

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program
- Simple objects: ”Strings”, Dates
- Streams, reading and writing to/from files, etc
- Exceptions
- Some typical Java beginner errors
- Collections of objects: lists and mappings

Java for the Internet

Objectives

- A little on concepts in object oriented programming
- Enough Java for this course
- Classes and objects
- Calling methods
- Statements, operators and primitive data types
- The mandatory ”Hello World” program
- Simple objects: ”Strings”, Dates
- Streams, reading and writing to/from files, etc
- Exceptions
- Some typical Java beginner errors
- Collections of objects: lists and mappings
- Threads, parallel execution

Object-oriented programming

Computer programming is the process (art?) of producing and maintaining source code for computer software. To this end one must use a programming language that can be either compiled or interpreted or both.

Object-oriented programming

Computer programming is the process (art?) of producing and maintaining source code for computer software. To this end one must use a programming language that can be either compiled or interpreted or both.

Different programming languages are good for different tasks and a good programming language for this kind of course should contain as much as possible of the basic building bricks for networking.

Object-oriented programming

Computer programming is the process (art?) of producing and maintaining source code for computer software. To this end one must use a programming language that can be either compiled or interpreted or both.

Different programming languages are good for different tasks and a good programming language for this kind of course should contain as much as possible of the basic building bricks for networking.

Therefore I have chosen Java.

Object-oriented programming

Computer programming is the process (art?) of producing and maintaining source code for computer software. To this end one must use a programming language that can be either compiled or interpreted or both.

Different programming languages are good for different tasks and a good programming language for this kind of course should contain as much as possible of the basic building bricks for networking.

Therefore I have chosen Java. It is made for programming applications divided into smaller programs that are spread over a network

Object-oriented programming

Computer programming is the process (art?) of producing and maintaining source code for computer software. To this end one must use a programming language that can be either compiled or interpreted or both.

Different programming languages are good for different tasks and a good programming language for this kind of course should contain as much as possible of the basic building bricks for networking.

Therefore I have chosen Java. It is made for programming applications divided into smaller programs that are spread over a network and that works by communicating over that network.

Classes and objects

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs
- Entities created from a Java class are called *objects*. An object created from a class is said to be an *instance* of that class.

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs
- Entities created from a Java class are called *objects*. An object created from a class is said to be an *instance* of that class.
- When creating an object, values may be sent as *parameters* in order to *initialize* the objects data.

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs
- Entities created from a Java class are called *objects*. An object created from a class is said to be an *instance* of that class.
- When creating an object, values may be sent as *parameters* in order to *initialize* the objects data.
- Thus, all objects that are created from a specific class have identical internal structure but values stored in the objects may differ.

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs
- Entities created from a Java class are called *objects*. An object created from a class is said to be an *instance* of that class.
- When creating an object, values may be sent as *parameters* in order to *initialize* the objects data.
- Thus, all objects that are created from a specific class have identical internal structure but values stored in the objects may differ.
- Data in the objects is stored in variables, called member variables. There are also class variables where you can store values that are to be shared by all instances of a class

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs
- Entities created from a Java class are called *objects*. An object created from a class is said to be an *instance* of that class.
- When creating an object, values may be sent as *parameters* in order to *initialize* the objects data.
- Thus, all objects that are created from a specific class have identical internal structure but values stored in the objects may differ.
- Data in the objects is stored in variables, called member variables. There are also class variables where you can store values that are to be shared by all instances of a class
- Variables of a certain class in Java are always pointers to an object of that class. They are called *references*

Classes and objects

- A class is like a template for constructing program entities containing both data and code to manipulate those data and are the basic building blocks for Java programs
- Entities created from a Java class are called *objects*. An object created from a class is said to be an *instance* of that class.
- When creating an object, values may be sent as *parameters* in order to *initialize* the objects data.
- Thus, all objects that are created from a specific class have identical internal structure but values stored in the objects may differ.
- Data in the objects is stored in variables, called member variables. There are also class variables where you can store values that are to be shared by all instances of a class
- Variables of a certain class in Java are always pointers to an object of that class. They are called *references*
- Un-initialized references have the value `null` (a reserved keyword in Java).

Classes and objects . . .

Classes and objects . . .

Example: date

Classes and objects . . .

Example: date

`java.util.Date` is a predefined class:

Classes and objects . . .

Example: date

`java.util.Date` is a predefined class:

```
import java.util.*;
```

Classes and objects . . .

Example: date

`java.util.Date` is a predefined class:

```
import java.util.*;
```

`New` is an operator in Java for constructing objects of all kinds

Classes and objects . . .

Example: date

`java.util.Date` is a predefined class:

```
import java.util.*;
```

`New` is an operator in Java for constructing objects of all kinds

```
Date d = new Date(2009,0,26);
```

0 for january!! But 26 for the 26:th day!!

Classes and objects . . .

Example: date

`java.util.Date` is a predefined class:

```
import java.util.*;
```

`New` is an operator in Java for constructing objects of all kinds

```
Date d = new Date(2009,0,26);
```

0 for january!! But 26 for the 26:th day!!

Destroy a date: `free(d);` (Automatic garbage collection)

Array

Array

- You can allocate arrays of variable size, of both primitive types and references to objects

Array

- You can allocate arrays of variable size, of both primitive types and references to objects
- All arrays are homogeneous (contain only objects of one type)

```
int[] aPrimitiveArray = new int[expression]
Date[] aDateArray = new Date[otherExpression]
```

Array

- You can allocate arrays of variable size, of both primitive types and references to objects
- All arrays are homogeneous (contain only objects of one type)

```
int[] aPrimitiveArray = new int[expression]
Date[] aDateArray = new Date[otherExpression]
```

- For arrays of references, the objects themselves are *not* instantiated

```
aDateArray[7] = new Date();  // today's date
```

Array

- You can allocate arrays of variable size, of both primitive types and references to objects
- All arrays are homogeneous (contain only objects of one type)

```
int[] aPrimitiveArray = new int[expression]
Date[] aDateArray = new Date[otherExpression]
```

- For arrays of references, the objects themselves are *not* instantiated

```
aDateArray[7] = new Date();  // today's date
```

- The first index is 0!

Array

- You can allocate arrays of variable size, of both primitive types and references to objects
- All arrays are homogeneous (contain only objects of one type)

```
int[] aPrimitiveArray = new int[expression]
Date[] aDateArray = new Date[otherExpression]
```

- For arrays of references, the objects themselves are *not* instantiated

```
aDateArray[7] = new Date();  // today's date
```

- The first index is 0!
- Arrays have a special ”field” called length

```
aDateArray.length
```

Array

- You can allocate arrays of variable size, of both primitive types and references to objects
- All arrays are homogeneous (contain only objects of one type)

```
int[] aPrimitiveArray = new int[expression]
Date[] aDateArray = new Date[otherExpression]
```

- For arrays of references, the objects themselves are *not* instantiated

```
aDateArray[7] = new Date();  // today's date
```

- The first index is 0!
- Arrays have a special ”field” called length

```
aDateArray.length
```

- Multidimensional arrays can be defined

```
int [][][] aPrimitiveTensor = new int[10][10][7];
```

Methods

Methods

- Besides *member variables* (of primitive types or references) Java classes also define *methods* that can manipulate the member variables

Methods

- Besides *member variables* (of primitive types or references) Java classes also define *methods* that can manipulate the member variables
- After constructing an object, you can call its methods:

```
Date d1 = new Date(2009,0,26);
Date d2 = new Date(2009,0,27);
boolean myTest = d1.before(d2);
```

Methods

- Besides *member variables* (of primitive types or references) Java classes also define *methods* that can manipulate the member variables
- After constructing an object, you can call its methods:

```
Date d1 = new Date(2009,0,26);
Date d2 = new Date(2009,0,27);
boolean myTest = d1.before(d2);
```

- Tip: If you use an object *only once* (to serve as a method argument) you don't need to give it a name. Create it directly!:

```
Date d1 = new Date(2009,0,26);
boolean myTest = d1.before(new Date(2009,0,27));
```

Methods . . .

Methods . . .

- You can define variables anywhere in a method, not just at the beginning

Methods . . .

- You can define variables anywhere in a method, not just at the beginning
- There can be more methods with the same name, as long as they have different number of arguments or different types of arguments:

```
void println(Object obj)
void println(int i)
void println(String s)
```

Methods . . .

- You can define variables anywhere in a method, not just at the beginning
- There can be more methods with the same name, as long as they have different number of arguments or different types of arguments:

```
void println(Object obj)
void println(int i)
void println(String s)
```

- Methods that have the same name as the class are used when building objects from the class. They are called *constructors*. We have already used one:

```
java.util.Date
public Date(int year, int month, int day)
```

This constructor is *deprecated* but that's not important for this course. It's good not to be surprised when the compiler warns you about it.

Java in this course

To use Java in the course you need to

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor
- Use objects already given to you as method arguments (e.g. `argv` in `main(String[] argv)` – later on)

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor
- Use objects already given to you as method arguments (e.g. `argv` in `main(String[] argv)` – later on)
- Invoke methods of these objects

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor
- Use objects already given to you as method arguments (e.g. `argv` in `main(String[] argv)` – later on)
- Invoke methods of these objects
- Pass the objects as arguments to constructors or methods of other objects.

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor
- Use objects already given to you as method arguments (e.g. `argv` in `main(String[] argv)` – later on)
- Invoke methods of these objects
- Pass the objects as arguments to constructors or methods of other objects.
- Identify the right class and constructor/method:

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor
- Use objects already given to you as method arguments (e.g. `argv` in `main(String[] argv)` – later on)
- Invoke methods of these objects
- Pass the objects as arguments to constructors or methods of other objects.
- Identify the right class and constructor/method:
  - from what you get at the lectures or

Java in this course

To use Java in the course you need to

- Create objects after identifying the needed class and constructor
- Use objects already given to you as method arguments (e.g. `argv` in `main(String[] argv)` – later on)
- Invoke methods of these objects
- Pass the objects as arguments to constructors or methods of other objects.
- Identify the right class and constructor/method:
  - from what you get at the lectures or
  - by browsing selected packages in the documentation
    `http://java.sun.com/j2se/1.5.0/docs/api/`
    or
    `http://java.sun.com/javase/6/docs/api/`

Java in this course . . . you need to . . .

Java in this course . . . you need to . . .

- use operators like
  `==, >=, <=, !=, &&, ||, &, |, ^, ...`

Java in this course . . . you need to . . .

- use operators like
  `==, >=, <=, !=, &&, ||, &, |, ^, ...`
- use loop constructions like
  `do, while, for`

Java in this course . . . you need to . . .

- use operators like
  `==, >=, <=, !=, &&, ||, &, |, ^, ...`
- use loop constructions like
  `do, while, for`
- use statements like
  `if/else, case/switch/default/break`

Java in this course . . . you need to . . .

- use operators like
  `==, >=, <=, !=, &&, ||, &, |, ^, ...`
- use loop constructions like
  `do, while, for`
- use statements like
  `if/else, case/switch/default/break`

They will be explained when we encounter them in the code

Public or not public

Public or not public

We never talked about the member variables of `java.util.Date` (only about constructors and methods)

Public or not public

We never talked about the member variables of `java.util.Date` (only about constructors and methods)

How to access these member variables? How to change them? Why are they not presented in the documentation?

Public or not public

We never talked about the member variables of `java.util.Date` (only about constructors and methods)

How to access these member variables? How to change them? Why are they not presented in the documentation?

Because you as a class user don't need to care about them. You can do everything you can think of with a Date by just calling its methods.

Public or not public

We never talked about the member variables of `java.util.Date` (only about constructors and methods)

How to access these member variables? How to change them? Why are they not presented in the documentation?

Because you as a class user don't need to care about them. You can do everything you can think of with a Date by just calling its methods.

By protecting access to members, classes ensure that programmers don't mess up with them, breaking something. Only public variables and methods are for "outsiders" (like me . . . and you) to use.

Public or not public . . .

For your curiosity, `Date` has a member variable, a `long`, when positive representing the number of milliseconds since Jan 1:st 1970 and when negative the number of milliseconds before Jan 1:st 1970

`static`

Java has no global variables but a class may define *class variables*
All variables declared `static` are allocated *only once* (and not for each object created)

`static`

Java has no global variables but a class may define *class variables*
All variables declared `static` are allocated *only once* (and not for each object created)

Thus,
```
public static int aGlobalVariable;
```
is not a good idea, as anyone can change the variable as they wish. Instead, declare the variable `final`.

`static`

Java has no global variables but a class may define *class variables*
All variables declared `static` are allocated *only once* (and not for each object created)

Thus,
`public static int aGlobalVariable;`
is not a good idea, as anyone can change the variable as they wish. Instead, declare the variable `final`.
Most of the `public static` variables that you will see are also `final`

`static`

Java has no global variables but a class may define *class variables*
All variables declared `static` are allocated *only once* (and not for each object created)

Thus,
`public static int aGlobalVariable;`
is not a good idea, as anyone can change the variable as they wish. Instead, declare the variable `final`.
Most of the `public static` variables that you will see are also `final`

Static methods can be defined to express procedures that are specific to the class but do not refer to any object.

static

Java has no global variables but a class may define *class variables*
All variables declared `static` are allocated *only once* (and not for each object created)

Thus,
`public static int aGlobalVariable;`
is not a good idea, as anyone can change the variable as they wish. Instead, declare the variable `final`.
Most of the `public static` variables that you will see are also `final`

Static methods can be defined to express procedures that are specific to the class but do not refer to any object.

You can use public static members by prefixing the name of the class

```
ClassName.staticField
ClassName.staticMethod(arguments)
```

Some comments on classes

Some comments on classes

- Encapsulation

Some comments on classes

- Encapsulation
  - Classes group related variables and methods together

Some comments on classes

- Encapsulation
  - Classes group related variables and methods together
  - Classes hide the way the methods process the member variables, exposing only the functionality their designers desire.

Some comments on classes

- Encapsulation
  - Classes group related variables and methods together
  - Classes hide the way the methods process the member variables, exposing only the functionality their designers desire.
- They can provide static variables and methods for other code to use without creating objects of that class

Some comments on classes

- Encapsulation
  - Classes group related variables and methods together
  - Classes hide the way the methods process the member variables, exposing only the functionality their designers desire.
- They can provide static variables and methods for other code to use without creating objects of that class
- The only way to write a statement is in a method.

Some comments on classes

- Encapsulation
  - Classes group related variables and methods together
  - Classes hide the way the methods process the member variables, exposing only the functionality their designers desire.
- They can provide static variables and methods for other code to use without creating objects of that class
- The only way to write a statement is in a method.
- The only way to have a method is to define a class.

Some comments on classes

- Encapsulation
  - Classes group related variables and methods together
  - Classes hide the way the methods process the member variables, exposing only the functionality their designers desire.
- They can provide static variables and methods for other code to use without creating objects of that class
- The only way to write a statement is in a method.
- The only way to have a method is to define a class.
- This is the only reason for which you'll write new classes in this course.

Finally – time to create the ”Hello World” program

The class is public, i.e. visible outside the file. The `.java` file name is always the same as the name of the class it encloses: HelloWorld.java

Finally – time to create the ”Hello World” program

The class is public, i.e. visible outside the file. The `.java` file name is always the same as the name of the class it encloses: HelloWorld.java

```
public class HelloWorld {
```

Finally – time to create the ”Hello World” program

The class is public, i.e. visible outside the file. The `.java` file name is always the same as the name of the class it encloses: HelloWorld.java

```
public class HelloWorld {
   public static void main(String[] argv) {
```

Finally – time to create the ”Hello World” program

The class is public, i.e. visible outside the file. The `.java` file name is always the same as the name of the class it encloses: HelloWorld.java

```
public class HelloWorld {
   public static void main(String[] argv) {
      System.out.println("Hello World!");
   }
}
```

Finally – time to create the ”Hello World” program

The class is public, i.e. visible outside the file. The `.java` file name is always the same as the name of the class it encloses: HelloWorld.java

```
public class HelloWorld {
   public static void main(String[] argv) {
      System.out.println("Hello World!");
   }
}
```

Compile with `javac HelloWorld.java`

Finally – time to create the ”Hello World” program

The class is public, i.e. visible outside the file. The `.java` file name is always the same as the name of the class it encloses: HelloWorld.java

```
public class HelloWorld {
   public static void main(String[] argv) {
      System.out.println("Hello World!");
   }
}
```

Compile with `javac HelloWorld.java`

Run by `java HelloWorld`

`public static void main(String[] argv)`

This is the entry point in all Java programs.

`public static void main(String[] argv)`

This is the entry point in all Java programs.

- `public`, to be visible from the outside

`public static void main(String[] argv)`

This is the entry point in all Java programs.

- `public`, to be visible from the outside
- `static`, because at the beginning of the program there are no objects created, so we need a method that can be accessed without having an object

`public static void main(String[] argv)`

This is the entry point in all Java programs.

- `public`, to be visible from the outside
- `static`, because at the beginning of the program there are no objects created, so we need a method that can be accessed without having an object
- `void` returns no value
  to return a value to the operating system, use `System.exit(value)`

`public static void main(String[] argv)`

This is the entry point in all Java programs.

- `public`, to be visible from the outside
- `static`, because at the beginning of the program there are no objects created, so we need a method that can be accessed without having an object
- `void` returns no value
  to return a value to the operating system, use `System.exit(value)`
- `String[] argv` (or any other name) is the list of arguments from the command line. The the number of arguments can be found out using `argv.length`

java.lang.System

Is a class containing methods that provide services related to the system where the program runs

java.lang.System

Is a class containing methods that provide services related to the system where the program runs

We didn't have to import the `java.lang.*` package to use `System`, because `java.lang` is always considered imported

java.lang.System

Is a class containing methods that provide services related to the system where the program runs

We didn't have to import the `java.lang.*` package to use `System`, because `java.lang` is always considered imported

`System.out` is a `public`, `static` and `final` variable of type `java.io.PrintStream`. Also, of course, `System.in` exists too.

`java.lang.System`

Is a class containing methods that provide services related to the system where the program runs

We didn't have to import the `java.lang.*` package to use `System`, because `java.lang` is always considered imported

`System.out` is a `public`, `static` and `final` variable of type `java.io.PrintStream`. Also, of course, `System.in` exists too.

```
static void exit(int status)
```

+ A number of other methods, not important for this course

java.lang.String

java.lang.String

- String

java.lang.String

- `String`
  - The way characters are represented in the `String` is hidden from the String user. Remember `Date`'s internal representation, and encapsulation

java.lang.String

- `String`
  - The way characters are represented in the `String` is hidden from the String user. Remember `Date`'s internal representation, and encapsulation
- Constants

java.lang.String

- `String`
  - The way characters are represented in the `String` is hidden from the String user. Remember `Date`'s internal representation, and encapsulation
- Constants
  - `System.out.println("Hello World!");`
    creates a `String` constant that is sent to `println` for printing on the screen

java.lang.String

- `String`
  - The way characters are represented in the `String` is hidden from the String user. Remember `Date`'s internal representation, and encapsulation
- Constants
  - `System.out.println("Hello World!");`
    creates a `String` constant that is sent to `println` for printing on the screen
  - We can declare a string by
    `String s = "Hello World";`

java.lang.String

- `String`
  - The way characters are represented in the `String` is hidden from the String user. Remember `Date`'s internal representation, and encapsulation
- Constants
  - `System.out.println("Hello World!");`
    creates a `String` constant that is sent to `println` for printing on the screen
  - We can declare a string by
    `String s = "Hello World";`
  - Java beginners often tend to do
    `s = new String("Hello World");`
    It's valid, but more resource-intensive.

String creation methods

- Indicating a double-quoted `"constant"` will create a String, `""` creates the empty String

String creation methods

- Indicating a double-quoted `"constant"` will create a String, `""` creates the empty String
- Operator + concatenates Strings, creating a new, longer String

```
String h = "Hello";
String w = "World";
String hw = h + " " + w;
```

String creation methods

- Indicating a double-quoted `"constant"` will create a String, `""` creates the empty String
- Operator + concatenates Strings, creating a new, longer String

```
String h = "Hello";
String w = "World";
String hw = h + " " + w;
```

- String creation from a portion of a char/byte array

String creation methods

- Indicating a double-quoted `"constant"` will create a String, `""` creates the empty String
- Operator + concatenates Strings, creating a new, longer String

```
String h = "Hello";
String w = "World";
String hw = h + " " + w;
```

- String creation from a portion of a char/byte array
  - `public String(char[] value, int offset, int count)` Constructor that builds a String of length 'count' starting from the 'offset' character of the 'value' char array

```
char[] h = 'h', 'e', 'l', 'l', 'o';
String s = new String(h, 0, 4);
System.out.println(s);     // will print "hell"
```

String creation methods

- Indicating a double-quoted `"constant"` will create a String, `""` creates the empty String
- Operator + concatenates Strings, creating a new, longer String

```
String h = "Hello";
String w = "World";
String hw = h + " " + w;
```

- String creation from a portion of a char/byte array
  - `public String(char[] value, int offset, int count)`
    Constructor that builds a String of length 'count' starting from the 'offset' character of the 'value' char array

```
char[] h = 'h', 'e', 'l', 'l', 'o';
String s = new String(h, 0, 4);
System.out.println(s);      // will print "hell"
```

  - `public String(char[] value)`
    does the same with `offset=0` and `count=value.length`

String creation methods

- Indicating a double-quoted `"constant"` will create a String, `""` creates the empty String
- Operator + concatenates Strings, creating a new, longer String

```
String h = "Hello";
String w = "World";
String hw = h + " " + w;
```

- String creation from a portion of a char/byte array
  - `public String(char[] value, int offset, int count)`
    Constructor that builds a String of length 'count' starting from the 'offset' character of the 'value' char array

```
char[] h = 'h', 'e', 'l', 'l', 'o';
String s = new String(h, 0, 4);
System.out.println(s);      // will print "hell"
```

  - `public String(char[] value)`
    does the same with `offset=0` and `count=value.length`
  - Many methods/constructors have such variants, it's important to understand the most general one

String examination

String examination

- `int length()`

String examination

- `int length()`
  `""`.`length()` returns 0
  `"Hello"`.`length()` returns 5

String examination

- `int length()`
  `""`.`length()` returns 0
  `"Hello"`.`length()` returns 5
- `char charAt(int index)`

String examination

- `int length()`
  `""`.`length()` returns 0
  `"Hello"`.`length()` returns 5
- `char charAt(int index)`
  `"Hello"`.`charAt(1)` returns ’`e`’ (first index is 0)

String examination

- `int length()`
  `""`.`length()` returns 0
  `"Hello"`.`length()` returns 5
- `char charAt(int index)`
  `"Hello"`.`charAt(1)` returns 'e' (first index is 0)
- `char[] toCharArray()` creates a new char array and copies all the string characters it

String examination

- `int length()`
  `""`.`length()` returns 0
  `"Hello"`.`length()` returns 5
- `char charAt(int index)`
  `"Hello"`.`charAt(1)` returns 'e' (first index is 0)
- `char[] toCharArray()` creates a new char array and copies all the string characters it
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  Copies chars from the String to the given `char[]array` beginning at `srcBegin` and ending at `srcEnd`
  It does not create a new `char array`, it needs an existing one (`dst`)

String examination

- `int length()`
  `"".length()` returns 0
  `"Hello".length()` returns 5
- `char charAt(int index)`
  `"Hello".charAt(1)` returns 'e' (first index is 0)
- `char[] toCharArray()` creates a new char array and copies all the string characters it
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`
  Copies chars from the String to the given `char[]array` beginning at `srcBegin` and ending at `srcEnd`
  It does not create a new `char array`, it needs an existing one (`dst`)
- Reflect on similarities between `String(char[] value, int offset, int length)` and `getChars(int start, int end, char[] dest, int destStart)`

String comparison

- `int compareTo(String anotherString)`
  returns -1 if the string is lexicographically smaller than `anotherString`, 0 if they are equal and +1 if it is larger than `anotherString`.

String comparison

- `int compareTo(String anotherString)`
  returns -1 if the string is lexicographically smaller than `anotherString`, 0 if they are equal and +1 if it is larger than `anotherString`.
- `boolean equals(Object obj)` returns 'true' for equality in value.
  Mark that with
  `String a = "x"; String b = "x";`
  `a==b` returns 'false' as '==' compares references
  `a==a` returns 'true' as a "points at" the same object
  `a.equals(b)` returns 'true' as you compare values.

String comparison

- `int compareTo(String anotherString)`
  returns -1 if the string is lexicographically smaller than `anotherString`, 0 if they are equal and +1 if it is larger than `anotherString`.
- `boolean equals(Object obj)` returns '`true`' for equality in value.
  Mark that with
  `String a = "x"; String b = "x";`
  `a==b` returns '`false`' as '`==`' compares references
  `a==a` returns '`true`' as a "points at" the same object
  `a.equals(b)` returns '`true`' as you compare values.
- `boolean startsWith(String prefix)`

String comparison

- `int compareTo(String anotherString)`
  returns -1 if the string is lexicographically smaller than `anotherString`, 0 if they are equal and +1 if it is larger than `anotherString`.
- `boolean equals(Object obj)` returns '`true`' for equality in value.
  Mark that with
  `String a = "x"; String b = "x";`
  `a==b` returns '`false`' as '`==`' compares references
  `a==a` returns '`true`' as a "points at" the same object
  `a.equals(b)` returns '`true`' as you compare values.
- `boolean startsWith(String prefix)`
- `boolean endsWith(String suffix)`

Searching strings

- `int indexOf(String what, int fromWhere)`

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)
- All returns either the index for the first found char or -1 to indicate ”not found”

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)
- All returns either the index for the first found char or -1 to indicate ”not found”
- Similar methods are available to look for a char (represented as an int)

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)
- All returns either the index for the first found char or -1 to indicate ”not found”
- Similar methods are available to look for a char (represented as an int)
- `"Hello".indexOf("he", 0)` returns 0

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)
- All returns either the index for the first found char or -1 to indicate ”not found”
- Similar methods are available to look for a char (represented as an int)
- `"Hello".indexOf("he", 0)` returns 0
- `"Hello".indexOf("he", 1)` returns -1

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)
- All returns either the index for the first found char or -1 to indicate ”not found”
- Similar methods are available to look for a char (represented as an int)
- `"Hello".indexOf("he", 0)` returns 0
- `"Hello".indexOf("he", 1)` returns -1
- `"Hello".indexOf("l", 1)` returns 2

Searching strings

- `int indexOf(String what, int fromWhere)`
- `int indexOf(String what)` (≡ `indexOf(what, 0)`)
- `int lastIndexOf(String what, int fromWhere)`
- `int lastIndexOf(String what)` (≡ `lastIndexOf(what, 0)`)
- All returns either the index for the first found char or -1 to indicate ”not found”
- Similar methods are available to look for a char (represented as an int)
- `"Hello".indexOf("he", 0)` returns 0
- `"Hello".indexOf("he", 1)` returns -1
- `"Hello".indexOf("l", 1)` returns 2
- `"Hello".lastIndexOf("l", 1)` returns 3

Other string operations

- `String concat(String s)`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`
- `"Hello".replace("l", "L") => "HeLLo"`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`
- `"Hello".replace("l", "L") => "HeLLo"`
- `"Hello".substring(2, 3) => "ll"`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`
- `"Hello".replace("l", "L") => "HeLLo"`
- `"Hello".substring(2, 3) => "ll"`
- `"   Hell o   ".trim() => "Hell o"`

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`
- `"Hello".replace("l", "L") => "HeLLo"`
- `"Hello".substring(2, 3) => "ll"`
- `"   Hell o   ".trim() => "Hell o"`
- These methods do not change the String but return new Strings

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`
- `"Hello".replace("l", "L") => "HeLLo"`
- `"Hello".substring(2, 3) => "ll"`
- `"   Hell o   ".trim() => "Hell o"`
- These methods do not change the String but return new Strings
- There are no methods that change Strings. Strings are immutable

Other string operations

- `String concat(String s)`
- `String toLowerCase()`
- `String toUpperCase()`
- `String replace(char old, char newc)`
- `String trim()`
- `String substring(int begin, int end)`
- `"Hello".toLowerCase() => "hello"`
- `"Hello".replace("l", "L") => "HeLLo"`
- `"Hello".substring(2, 3) => "ll"`
- `"   Hell o   ".trim() => "Hell o"`
- These methods do not change the String but return new Strings
- There are no methods that change Strings. Strings are immutable
- If you really want changeable strings, use StringBuffer

Garbage Collection

- Objects that are not referenced from anywhere are garbage collected

Garbage Collection

- Objects that are not referenced from anywhere are garbage collected
  - When a reference is made null:

```
String s = "Hello World";
System.out.println(s);
s = null;
```

Garbage Collection

- Objects that are not referenced from anywhere are garbage collected
  - When a reference is made null:

```
String s = "Hello World";
System.out.println(s);
s = null;
```

  - When a method that has a variable referring to the object ends:

```
void myMethod {
    String s = "Hello World";
    System.out.println(s);
}
```

Garbage Collection

- Objects that are not referenced from anywhere are garbage collected
  - When a reference is made null:
    ```
    String s = "Hello World";
    System.out.println(s);
    s = null;
    ```
  - When a method that has a variable referring to the object ends:
    ```
    void myMethod {
        String s = "Hello World";
        System.out.println(s);
    }
    ```
  - When the (only) referrer object is garbage collected:
    ```
    class MyClass { String s; ...}
    ... MyClass mc = new MyClass(); ...
    ```
    when `mc` dies, `s` will be garbage collected unless `mc` passes the reference `s` to some other object.

Inheritance

- `java.lang.String` *`extends`* `java.lang.Object`

Inheritance

- `java.lang.String` *`extends`* `java.lang.Object`
- Thus every instance of `String` is also an instance of `Object` but no all instances of `Object` are instances of `String`.

```
String s = "Hello World";
Object obj = s; // OK! s == obj => true
s = obj; // Error! Not all objects are strings
```

Inheritance

- `java.lang.String` *`extends`* `java.lang.Object`
- Thus every instance of `String` is also an instance of `Object` but no all instances of `Object` are instances of `String`.

```
String s = "Hello World";
Object obj = s; // OK! s == obj => true
s = obj; // Error! Not all objects are strings
```

- `String` is a *subclass* to `Object` and `Object` is a *superclass* to `String`

Inheritance

- `java.lang.String` *`extends`* `java.lang.Object`
- Thus every instance of `String` is also an instance of `Object` but no all instances of `Object` are instances of `String`.

```
String s = "Hello World";
Object obj = s; // OK! s == obj => true
s = obj; // Error! Not all objects are strings
```

- `String` is a *subclass* to `Object` and `Object` is a *superclass* to `String`
- Subclasses *inherit* all member variables and methods to from their superclasses and may override some, and may add new ones. `equals(Object what)` is actually a method that `String` inherits from `Object`. One can test equality for any kind of Java object not just for strings. String has to override `equals()` to define equality for Strings On the other hand, you can't `trim()` just any kind of Object, `trim()` only makes sense for strings. So it's a method that `String` adds.

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects
- `String toString()`

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass. So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects
- `String toString()`
  - Many classes redefine `toString()` to describe the object state in human-readable form for debugging purposes.

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects
- `String toString()`
  - Many classes redefine `toString()` to describe the object state in human-readable form for debugging purposes.
  - `toString()` is automatically called on non-`String` objects by the `+` operator

    ```
    System.out.println("Hello world, it is "+ new Date());
    ```

    // prints the message and today's date and time.
    Note that `"Hello World, I work for "+(1+1)+"dollars a day"` will evaluate the `int` expression and include it in the message
    That's why `println()` doesn't need formatting as output in most languages!

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects
- `String toString()`
  - Many classes redefine `toString()` to describe the object state in human-readable form for debugging purposes.
  - `toString()` is automatically called on non-`String` objects by the `+` operator

    ```
    System.out.println("Hello world, it is "+ new Date());
    ```

    // prints the message and today's date and time.
    Note that `"Hello World, I work for "+(1+1)+"dollars a day"` will evaluate the `int` expression and include it in the message
    That's why `println()` doesn't need formatting as output in most languages!
  - `Class getClass()` important when you debug and are not sure about the actual type of an object.

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects
- `String toString()`
  - Many classes redefine `toString()` to describe the object state in human-readable form for debugging purposes.
  - `toString()` is automatically called on non-`String` objects by the `+` operator

    ```
    System.out.println("Hello world, it is "+ new Date());
    ```

    // prints the message and today's date and time.
    Note that `"Hello World, I work for "+(1+1)+"dollars a day"` will evaluate the `int` expression and include it in the message
    That's why `println()` doesn't need formatting as output in most languages!
  - `Class getClass()` important when you debug and are not sure about the actual type of an object.
  - `System.out.print(myObj.getClass().getName());`

Some `java.lang.Object` methods

- If a class has no superclass indicated, `Object` is automatically its superclass.
  So all `Object` methods are present in all classes.
- The rule about `==` and `equals()` stands for all objects
- `String toString()`
  - Many classes redefine `toString()` to describe the object state in human-readable form for debugging purposes.
  - `toString()` is automatically called on non-`String` objects by the `+` operator

    ```
    System.out.println("Hello world, it is "+ new Date());
    ```

    // prints the message and today's date and time.
    Note that `"Hello World, I work for "+(1+1)+"dollars a day"` will evaluate the `int` expression and include it in the message
    That's why `println()` doesn't need formatting as output in most languages!
  - `Class getClass()` important when you debug and are not sure about the actual type of an object.
  - `System.out.print(myObj.getClass().getName());`
  - You can also check if the object is from a certain class using the `instanceof` operator: `obj instanceof java.lang.String`

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

A: Look also in the documentation of the superclass (Object in this case).

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

A: Look also in the documentation of the superclass (Object in this case).

- Only added methods (like `trim()`) and overridden methods (like `equals()`) are shown in the method table of the documentation.

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

A: Look also in the documentation of the superclass (Object in this case).

- Only added methods (like `trim()`) and overridden methods (like `equals()`) are shown in the method table of the documentation.
- Since any `String` is also an `Object`, you can safely call `getClass()` for your `String`.

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

A: Look also in the documentation of the superclass (Object in this case).

- Only added methods (like `trim()`) and overridden methods (like `equals()`) are shown in the method table of the documentation.
- Since any `String` is also an `Object`, you can safely call `getClass()` for your `String`.
- Nowadays such hard-to-find methods are shown in the ”methods inherited from . . . ” documentation section.

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

A: Look also in the documentation of the superclass (Object in this case).

- Only added methods (like `trim()`) and overridden methods (like `equals()`) are shown in the method table of the documentation.
- Since any `String` is also an `Object`, you can safely call `getClass()` for your `String`.
- Nowadays such hard-to-find methods are shown in the ”methods inherited from . . . ” documentation section.
- This problem is bigger when you actually don’t know exactly what method to call. You might miss the right method because it’s in the superclass.

”Q&A” on ”missing methods”

Q: I’m supposed to call method `getClass()` of a `String`, but I can’t see it in the `String` class documentation

A: Look also in the documentation of the superclass (Object in this case).

- Only added methods (like `trim()`) and overridden methods (like `equals()`) are shown in the method table of the documentation.
- Since any `String` is also an `Object`, you can safely call `getClass()` for your `String`.
- Nowadays such hard-to-find methods are shown in the ”methods inherited from . . . ” documentation section.
- This problem is bigger when you actually don’t know exactly what method to call. You might miss the right method because it’s in the superclass.
- So get used to look at “methods inherited from” and at the superclass itself.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.
- `print` and `println` differs in that `println` adds a newline at the end of the printed string.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.
- `print` and `println` differs in that `println` adds a newline at the end of the printed string.
- `println()` with no arguments just prints a newline.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.
- `print` and `println` differs in that `println` adds a newline at the end of the printed string.
- `println()` with no arguments just prints a newline.
- Streams often use internal buffers, so not everything you print is actually sent immediately. `void flush()` orders such a sending.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.
- `print` and `println` differs in that `println` adds a newline at the end of the printed string.
- `println()` with no arguments just prints a newline.
- Streams often use internal buffers, so not everything you print is actually sent immediately. `void flush()` orders such a sending.
- `void close()` gives back all resources to the system, the stream is unusable afterwards. You should do this when you are done with a stream that you created (but *never* for `System.out`).

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.
- `print` and `println` differs in that `println` adds a newline at the end of the printed string.
- `println()` with no arguments just prints a newline.
- Streams often use internal buffers, so not everything you print is actually sent immediately. `void flush()` orders such a sending.
- `void close()` gives back all resources to the system, the stream is unusable afterwards. You should do this when you are done with a stream that you created (but *never* for `System.out`).
- `close()` will typically `flush()` the stream before closing it.

java.io.PrintStream

- Mostly for debugging purposes. `System.out` is the most famous `PrintStream`.
- Has `print()` and `println()` methodes for all primitive types.
- `void print(Object obj)` prints ”`null`” if `obj` is ”null” and `obj.toString()` otherwise.
- `print` and `println` differs in that `println` adds a newline at the end of the printed string.
- `println()` with no arguments just prints a newline.
- Streams often use internal buffers, so not everything you print is actually sent immediately. `void flush()` orders such a sending.
- `void close()` gives back all resources to the system, the stream is unusable afterwards. You should do this when you are done with a stream that you created (but *never* for `System.out`).
- `close()` will typically `flush()` the stream before closing it.
- Creation: `PrintStream(java.io.OutputStream out)`

”Q&A” on `java.io.PrintStream`

Q: I want to create a `PrintStream` to print stuff to a file. Since the `PrintStream(OutputStream)` constructor needs an `OutputStream`, I want to create an `OutputStream` but the compiler tells me that I can’t, as `OutputStream` is an abstract class, whatever that means.

”Q&A” on `java.io.PrintStream`

Q: I want to create a `PrintStream` to print stuff to a file. Since the `PrintStream(OutputStream)` constructor needs an `OutputStream`, I want to create an `OutputStream` but the compiler tells me that I can’t, as `OutputStream` is an abstract class, whatever that means.

A: You can use an instance of any `OutputStream` subclass. In your case, you need a `java.io.FileOutputStream`. So if you see a type T in a constructor/method signature, it actually means “type T and all its subclasses”.

”Q&A” on `java.io.PrintStream`

Q: I want to create a `PrintStream` to print stuff to a file. Since the `PrintStream(OutputStream)` constructor needs an `OutputStream`, I want to create an `OutputStream` but the compiler tells me that I can’t, as `OutputStream` is an abstract class, whatever that means.

A: You can use an instance of any `OutputStream` subclass. In your case, you need a `java.io.FileOutputStream`. So if you see a type T in a constructor/method signature, it actually means “type T and all its subclasses”.

We will talk about abstract classes shortly

Write to a file with `PrintStream`

This program writes ”Hello world!” to a file indicated as first argument
Run it as `java WriteToFile1 filename`

```
import java.io.*;
public class WriteToFile1 {
   public static void main(String[] argv) {
      OutputStream file = null;
      try {
          file = new FileOutputStream(argv[0]);
      }
      catch (FileNotFoundException fnf) {
         System.err.println("File not found: " + argv[0]);
         fnf.printStackTrace(); // for debugging
         System.exit(1);
      }
      PrintStream ps = new PrintStream(file);
      ps.println("Hello world!");
      ps.close();
   }
}
```

Exceptions

In Java you distinguish operational code from error recovery code so when something goes wrong an *exception* is thrown.

Exceptions

In Java you distinguish operational code from error recovery code so when something goes wrong an *exception* is thrown.

That's why `PrintStream` doesn't return any values. Instead an *`Exception`* is thrown of a type and with a name that gives more meaning to and more information about the error.

Exceptions

In Java you distinguish operational code from error recovery code so when something goes wrong an *exception* is thrown.

That's why `PrintStream` doesn't return any values. Instead an *`Exception`* is thrown of a type and with a name that gives more meaning to and more information about the error.

The `FileOutputStream` constructor which we used is declared like this:

```
FileOutputStream(String filename) throws java.io.FileNotFoundException;
```

Exceptions

In Java you distinguish operational code from error recovery code so when something goes wrong an *exception* is thrown.

That's why `PrintStream` doesn't return any values. Instead an *`Exception`* is thrown of a type and with a name that gives more meaning to and more information about the error.

The `FileOutputStream` constructor which we used is declared like this:

```
FileOutputStream(String filename) throws java.io.FileNotFoundException;
```

If the String we used is not a valid file name, the `catch` block is executed.

Exceptions

In Java you distinguish operational code from error recovery code so when something goes wrong an *exception* is thrown.

That's why `PrintStream` doesn't return any values. Instead an *`Exception`* is thrown of a type and with a name that gives more meaning to and more information about the error.

The `FileOutputStream` constructor which we used is declared like this:

```
FileOutputStream(String filename) throws java.io.FileNotFoundException;
```

If the String we used is not a valid file name, the `catch` block is executed.

That way, Java separates ”normal code” from ”code that executes when things go wrong”.

Exceptions

In Java you distinguish operational code from error recovery code so when something goes wrong an *exception* is thrown.

That's why `PrintStream` doesn't return any values. Instead an *`Exception`* is thrown of a type and with a name that gives more meaning to and more information about the error.

The `FileOutputStream` constructor which we used is declared like this:

```
FileOutputStream(String filename) throws java.io.FileNotFoundException;
```

If the String we used is not a valid file name, the `catch` block is executed.

That way, Java separates ”normal code” from ”code that executes when things go wrong”.

An exception is an object like any other, has a class, . . .

Treating exceptions

- If we don't treat the exception, the compiler will punish us

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)
- To treat an exception we can either catch the exception

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)
- To treat an exception we can either catch the exception
  - Put one or more methods in a `try... catch(...)...` block and constructors that are declared as throwing exceptions

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)
- To treat an exception we can either catch the exception
  - Put one or more methods in a `try... catch(...)...` block and constructors that are declared as throwing exceptions
  - You should `catch()` either that exception class, or one of its superclasses

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)
- To treat an exception we can either catch the exception
  - Put one or more methods in a `try... catch(...)...` block and constructors that are declared as throwing exceptions
  - You should `catch()` either that exception class, or one of its superclasses
  - `catch(IOException exc)` would have worked as well because `java.io.FileNotFoundException` is a subclass of `java.io.IOException`

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)
- To treat an exception we can either catch the exception
  - Put one or more methods in a `try... catch(...)...` block and constructors that are declared as throwing exceptions
  - You should `catch()` either that exception class, or one of its superclasses
  - `catch(IOException exc)` would have worked as well because `java.io.FileNotFoundException` is a subclass of `java.io.IOException`
  - You can have more than one `catch()` block for a `try`

Treating exceptions

- If we don't treat the exception, the compiler will punish us
- No need to treat subclasses of `java.lang.RuntimeException` (typically programmer errors) or `java.lang.Error` (abnormal condition)
- To treat an exception we can either catch the exception
  - Put one or more methods in a `try... catch(...)...` block and constructors that are declared as throwing exceptions
  - You should `catch()` either that exception class, or one of its superclasses
  - `catch(IOException exc)` would have worked as well because `java.io.FileNotFoundException` is a subclass of `java.io.IOException`
  - You can have more than one `catch()` block for a `try`
- Or we can throw the exception further from the method if we write `public static void main(String argv[]) throws IOException`

Exceptions . . .

This program writes ”Hello world!” to a file indicated as first argument
Run it as `java WriteToFile2 file.txt`

```
import java.io.*;
public class WriteToFile2 {
   public static void main(String[] argv) throws IOException {
      PrintStream ps = new PrintStream(new FileOutputStream(argv[0]));
      ps.println("Hello world!");
      ps.close();
   }
}
```

Important `RuntimeExceptions`

- `java.lang.NullPointerException`

Important `RuntimeExceptions`

- `java.lang.NullPointerException`
  - occurs when you try to access a member (method or field) of a null reference

```
Object obj = null;
String s = obj.toString(); // exception!
```

Important `RuntimeExceptions`

- `java.lang.NullPointerException`
  - occurs when you try to access a member (method or field) of a null reference

```
Object obj = null;
String s = obj.toString(); // exception!
```

  - It's good to check if you are not sure

```
if(obj != null) s = obj.toString();
```

Important `RuntimeExceptions`

- `java.lang.NullPointerException`
  - occurs when you try to access a member (method or field) of a null reference

    ```
    Object obj = null;
    String s = obj.toString(); // exception!
    ```
  - It's good to check if you are not sure

    ```
    if(obj != null) s = obj.toString();
    ```
- `java.lang.ArrayIndexOutOfBoundsException` occurs when you try to access an array element with an index that's negative or larger than the array size

  ```
  char[] arr= new char[20]; char exc=arr[40];
  ```

Important `RuntimeExceptions`

- `java.lang.NullPointerException`
  - occurs when you try to access a member (method or field) of a null reference

    ```
    Object obj = null;
    String s = obj.toString(); // exception!
    ```
  - It's good to check if you are not sure

    ```
    if(obj != null) s = obj.toString();
    ```
- `java.lang.ArrayIndexOutOfBoundsException` occurs when you try to access an array element with an index that's negative or larger than the array size

  ```
  char[] arr= new char[20]; char exc=arr[40];
  ```
- `java.lang.StringIndexOutOfBoundsException` occurs when you use an illegal (negative or too large) index when calling `charAt()`, `indexOf()`, `substring()`,...
  Typical: `indexOf()` returns `-1` (not found) and you use that index for `substring()` without checking it

Important `RuntimeExceptions`

- `java.lang.NullPointerException`
  - occurs when you try to access a member (method or field) of a null reference

    ```
    Object obj = null;
    String s = obj.toString(); // exception!
    ```
  - It's good to check if you are not sure

    ```
    if(obj != null) s = obj.toString();
    ```
- `java.lang.ArrayIndexOutOfBoundsException` occurs when you try to access an array element with an index that's negative or larger than the array size

  ```
  char[] arr= new char[20]; char exc=arr[40];
  ```
- `java.lang.StringIndexOutOfBoundsException` occurs when you use an illegal (negative or too large) index when calling `charAt()`, `indexOf()`, `substring()`,...
  Typical: `indexOf()` returns `-1` (not found) and you use that index for `substring()` without checking it
- `java.lang.SecurityException`
  some operations are forbidden for some of the Java code

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection
- You can use any of these to create a `PrintStream`.
  That's a case when the OutputStream generalization is useful!

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection
- You can use any of these to create a `PrintStream`.
  That's a case when the OutputStream generalization is useful!
- `PrintStream` is actually an `OutputStream` itself

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection
- You can use any of these to create a `PrintStream`.
  That's a case when the OutputStream generalization is useful!
- `PrintStream` is actually an `OutputStream` itself
- `flush()` and `close()` are actually `OutputStream` methods.

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection
- You can use any of these to create a `PrintStream`.
  That's a case when the OutputStream generalization is useful!
- `PrintStream` is actually an `OutputStream` itself
- `flush()` and `close()` are actually `OutputStream` methods.
  - Indeed, they make sense for all streams, not only for `OutputStream`

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection
- You can use any of these to create a `PrintStream`.
  That's a case when the OutputStream generalization is useful!
- `PrintStream` is actually an `OutputStream` itself
- `flush()` and `close()` are actually `OutputStream` methods.
  - Indeed, they make sense for all streams, not only for `OutputStream`
  - The difference is that the `OutputStream` declares them as throwing `IOException`.

Abstract classes

`java.io.OutputStream`

- Abstract classes are used to express general concepts that are useful as generalization but make no sense to construct objects.
- `java.io.OutputStream` is a generic output stream. Some subclasses:
  - Writing to a file: `java.io.FileOutputStream`
  - Writing to a byte array: `java.io.ByteArrayOutputStream`
  - Later on we will see streams that write to a TCP connection
- You can use any of these to create a `PrintStream`.
  That's a case when the OutputStream generalization is useful!
- `PrintStream` is actually an `OutputStream` itself
- `flush()` and `close()` are actually `OutputStream` methods.
  - Indeed, they make sense for all streams, not only for `OutputStream`
  - The difference is that the `OutputStream` declares them as throwing `IOException`.
  - No method of `PrintStream` throws `IOException` in order to make life easier for the programmer when debugging with `PrintStream`

Writing to a file using `OutputStream`'s write()

- The fundamental writing method is:

```
void write(byte[] value, int offset, int length)
throws IOException
```

Writing to a file using `OutputStream`'s write()

- The fundamental writing method is:
  `void write(byte[] value, int offset, int length) throws IOException`
- This program writes to a file indicated as first argument a message indicated as it's second argument:
  Run with `java WriteToFile3 filename "blah blah"`

```
import java.io.*;
public class WriteToFile3 {
   public static void main(String[] argv)
      throws IOException {
      OutputStream file = new FileOutputStream(argv[0]);
      String msg = argv[1];
      file.write(msg.getBytes(), 0, msg.length());
      file.close();
   }
}
```

Buffered writing

- If we do lots of `write()` operations with short strings like we had, performance will suffer because each `file.write()` will generate a disk (or network) access

Buffered writing

- If we do lots of `write()` operations with short strings like we had, performance will suffer because each `file.write()` will generate a disk (or network) access
- This is also a problem when writing to the Internet

Buffered writing

- If we do lots of `write()` operations with short strings like we had, performance will suffer because each `file.write()` will generate a disk (or network) access
- This is also a problem when writing to the Internet
- Buffered streams help to avoid this by gathering info sent to more `write()` into a byte array called *buffer*.
  When the buffer gets full, its content is sent further (to the disk in this case).

Buffered writing

- If we do lots of `write()` operations with short strings like we had, performance will suffer because each `file.write()` will generate a disk (or network) access
- This is also a problem when writing to the Internet
- Buffered streams help to avoid this by gathering info sent to more `write()` into a byte array called *buffer*.
  When the buffer gets full, its content is sent further (to the disk in this case).
- To empty the buffer before it gets full, you can call the `flush()` method

Buffered writing

- If we do lots of `write()` operations with short strings like we had, performance will suffer because each `file.write()` will generate a disk (or network) access
- This is also a problem when writing to the Internet
- Buffered streams help to avoid this by gathering info sent to more `write()` into a byte array called *buffer*.
  When the buffer gets full, its content is sent further (to the disk in this case).
- To empty the buffer before it gets full, you can call the `flush()` method
- As for `PrintStream` you need an `OutputStream` when you create a `BufferedOutputStream`. When writing to a file, `FileOutputStream` will be the choise.

Adding buffering to streaming code

- `OutputStream file = new FileOutputStream(argv[0]);`
  is changed to
  ```
  OutputStream file = new BufferedOutputStream(
    new FileOutputStream(argv[0]));
  ```

Adding buffering to streaming code

- `OutputStream file = new FileOutputStream(argv[0]);`
  is changed to
```
OutputStream file = new BufferedOutputStream(
  new FileOutputStream(argv[0]));
```
- We can also buffer the `PrintStream` we created
```
PrintStream ps = new PrintStream(new
FileOutputStream(argv[0]));
```
  is changed to
```
PrintStream ps = new PrintStream(
  new BufferedOutputStream(
  new FileOutputStream(argv[0])));
```

Adding buffering to streaming code

- `OutputStream file = new FileOutputStream(argv[0]);`
  is changed to
  ```
  OutputStream file = new BufferedOutputStream(
    new FileOutputStream(argv[0]));
  ```
- We can also buffer the `PrintStream` we created
  ```
  PrintStream ps = new PrintStream(new
  FileOutputStream(argv[0]));
  ```
  is changed to
  ```
  PrintStream ps = new PrintStream(
    new BufferedOutputStream(
    new FileOutputStream(argv[0])));
  ```
- The rest of the programs stays the same!

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

```
int read(byte[] where, int offset, int length) throws IOException
```

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

```
int read(byte[] where, int offset, int length) throws IOException
```

  - Reads at most `length` bytes

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

```
int read(byte[] where, int offset, int length) throws IOException
```

  - Reads at most `length` bytes
  - Puts the read bytes in the `where` array, starting from the `offset` position

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

```
int read(byte[] where, int offset, int length) throws IOException
```

  - Reads at most `length` bytes
  - Puts the read bytes in the `where` array, starting from the `offset` position
  - If there are no bytes available right now, `read()` blocks until more bytes come, or the end of data is signaled.

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

```
int read(byte[] where, int offset, int length) throws IOException
```

  - Reads at most `length` bytes
  - Puts the read bytes in the `where` array, starting from the `offset` position
  - If there are no bytes available right now, `read()` blocks until more bytes come, or the end of data is signaled.
  - Returns the number of bytes read or -1 if there is no more data

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

  ```
  int read(byte[] where, int offset, int length) throws IOException
  ```

  - Reads at most `length` bytes
  - Puts the read bytes in the `where` array, starting from the `offset` position
  - If there are no bytes available right now, `read()` blocks until more bytes come, or the end of data is signaled.
  - Returns the number of bytes read or -1 if there is no more data
- `int available()` returns the number of bytes currently available to be read

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

  ```
  int read(byte[] where, int offset, int length) throws IOException
  ```

  - Reads at most `length` bytes
  - Puts the read bytes in the `where` array, starting from the `offset` position
  - If there are no bytes available right now, `read()` blocks until more bytes come, or the end of data is signaled.
  - Returns the number of bytes read or -1 if there is no more data
- `int available()` returns the number of bytes currently available to be read
- `void close()` gives up the resources used by the stream, as for output streams

Reading, `InputStream`

- `java.io.InputStream` is the abstract superclass
- Fundamental reading method:

  ```
  int read(byte[] where, int offset, int length) throws IOException
  ```

  - Reads at most `length` bytes
  - Puts the read bytes in the `where` array, starting from the `offset` position
  - If there are no bytes available right now, `read()` blocks until more bytes come, or the end of data is signaled.
  - Returns the number of bytes read or -1 if there is no more data
- `int available()` returns the number of bytes currently available to be read
- `void close()` gives up the resources used by the stream, as for output streams
- Similar to output streams, there exist a variety of input streams `FileInputStream`, `ByteArrayInputStream`, `BufferedInputStream`, ...

Showing the content of a file

This program prints the content of a file indicated as first argument
Run with `java ReadFromFile filnamn`

```
import java.io.*;
public class ReadFromFile {
   public static void main (String[] argv) throws IOException {
   InputStream file =
      new BufferedInputStream(new FileInputStream(argv[0]));
   byte[] buffer = new byte[1024]; // 1 kB buffer
   int n;
   while ((n = file.read(buffer, 0, 1024)) != -1)
   // called for each kB of file content
       System.out.write(buffer, 0, n);
   // System.out is both a PrintStream and an OutputStream
   file.close();
   System.out.flush();
   // Not necessarily needed, just to make sure that the
   // content shows on the screen immediately.
   }
}
```

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte $\leftrightarrow$ char translation isn't possible, since chars require 2 bytes in some alphabets.

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte $\leftrightarrow$ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte $\leftrightarrow$ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).
- `Writer`s are similar to `OutputStream`s

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte ↔ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).
- `Writer`s are similar to `OutputStream`s
  - but their `write()` method has a `char[]` argument instead of `byte[]`

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte ↔ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).
- `Writer`s are similar to `OutputStream`s
  - but their `write()` method has a `char[]` argument instead of `byte[]`
  - To make a `Writer` out of an `OutputStream` use `OutputStreamWriter`. There is no reverse operation!

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte ↔ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).
- `Writer`s are similar to `OutputStream`s
  - but their `write()` method has a `char[]` argument instead of `byte[]`
  - To make a `Writer` out of an `OutputStream` use `OutputStreamWriter`. There is no reverse operation!
  - `PrintStream` should have been `PrintWriter` from start as it mostly carries text.

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte ↔ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).
- `Writer`s are similar to `OutputStream`s
  - but their `write()` method has a `char[]` argument instead of `byte[]`
  - To make a `Writer` out of an `OutputStream` use `OutputStreamWriter`. There is no reverse operation!
  - `PrintStream` should have been `PrintWriter` from start as it mostly carries text.
  - `FileWriter`, `BufferedWriter` and `CharArrayWriter` have their `OutputStream` correspondents `StringWriter` is new, writes content to a String

`java.io.Reader` and `java.io.Writer`

- Already in java 1.1 they realized that byte ↔ char translation isn't possible, since chars require 2 bytes in some alphabets.
- You should use `Reader`s and `Writer`s instead of `InputStream`s and `OutputStream`s when you know that all transferred content will be text and not binary data (e.g. jpeg images).
- `Writer`s are similar to `OutputStream`s
  - but their `write()` method has a `char[]` argument instead of `byte[]`
  - To make a `Writer` out of an `OutputStream` use `OutputStreamWriter`. There is no reverse operation!
  - `PrintStream` should have been `PrintWriter` from start as it mostly carries text.
  - `FileWriter`, `BufferedWriter` and `CharArrayWriter` have their `OutputStream` correspondents `StringWriter` is new, writes content to a String
- `Reader`s are similar to `InputStream`s but their `read()` method has a `char[]` argument instead of `byte[]`.
  To make a `Reader` out of an `InputStream` use `InputStreamReader`
  `BufferedReader` is like `ByteArrayInputStream` but has a `readLine()` method to read line by line. Returns `null` after the last line.

Showing the content of a text file

This program prints the content of a text file indicated as argument.
Run with `java ReadFromTextFile filnamn`

```
import java.io.*;
public class ReadFromTextFile {
  public static void main (String[] argv) throws IOException {
    Reader file = new BufferedReader(new FileReader(argv[0]));
    char[] buffer = new char[1024]; // 1 kB
    int n;
    while ((n = file.read(buffer, 0, 1024)) != -1)
      System.out.print(new String(buffer, 0, n));
    file.close();
    System.out.flush();
  }
}
```

If you (really) want to use `write()`

This program prints the content of a text file indicated as argument

```
import java.io.*;
public class ReadFromTextFile2 {
   public static void main (String[] argv)  throws IOException {
      Reader file =
         new BufferedReader(new FileReader(argv[0]));
      Writer sysout =
         new BufferedWriter(new OutputStreamWriter(System.out));
      // Transform System.out to a Writer!!!
      char[] buffer = new char[1024]; // 1 kB
      int n;
      while ((n = file.read(buffer, 0, 1024)) != -1)
         sysout.write(buffer, 0, n);
      file.close();
      sysout.flush();
   }
}
```

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects
- `Object get(int index)` returns a reference to the object at position `index`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects
- `Object get(int index)` returns a reference to the object at position `index`
- `boolean contains(Object obj)` checks whether there is an object that `equals(obj)`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects
- `Object get(int index)` returns a reference to the object at position `index`
- `boolean contains(Object obj)` checks whether there is an object that `equals(obj)`
- `int indexOf(Object obj)` returns the index for the first object that `equals(obj)`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects
- `Object get(int index)` returns a reference to the object at position `index`
- `boolean contains(Object obj)` checks whether there is an object that `equals(obj)`
- `int indexOf(Object obj)` returns the index for the first object that `equals(obj)`
- `int lastIndexOf(Object obj)` returns the index for the last object that `equals(obj)`

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects
- `Object get(int index)` returns a reference to the object at position `index`
- `boolean contains(Object obj)` checks whether there is an object that `equals(obj)`
- `int indexOf(Object obj)` returns the index for the first object that `equals(obj)`
- `int lastIndexOf(Object obj)` returns the index for the last object that `equals(obj)`
- `int size()` returns the number of objects present

Object collections: Lists

- `java.util.ArrayList` is an array of `Object`s
- Differs from `Object[]` as it grows according to needs and has an initial capacity.
- `void add(int index, Object element)` adds `element` at position `index`
- `boolean add(Object element)` adds `element` at the end
- `void set(int index, Object element)` replaces the object at position `index`
- `void remove(int index)` removes the object at position `index`
- `void remove(Object obj)` removes the first object that `equals(obj)`
- `void clear()` removes all objects
- `Object get(int index)` returns a reference to the object at position `index`
- `boolean contains(Object obj)` checks whether there is an object that `equals(obj)`
- `int indexOf(Object obj)` returns the index for the first object that `equals(obj)`
- `int lastIndexOf(Object obj)` returns the index for the last object that `equals(obj)`
- `int size()` returns the number of objects present
- modern choise instead of older class `java.util.Vector`

”Q&A” on Object

Q: I want to add a `String` object to my `ArrayList`, but the documentation says that I have to add an `Object` instance.

”Q&A” on Object

Q: I want to add a `String` object to my `ArrayList`, but the documentation says that I have to add an `Object` instance.

A: All `String` objects are `Object`s as well, so go ahead and add your string

”Q&A” on Object

Q: I want to add a `String` object to my `ArrayList`, but the documentation says that I have to add an `Object` instance.

A: All `String` objects are `Object`s as well, so go ahead and add your string

Q: I know i added a `String`-object to my `ArrayList` but the `get()` method returns an `Object` object. What to do? I want to call `substring()` but there’s no `substring()` method in `Object`

”Q&A” on Object

Q: I want to add a `String` object to my `ArrayList`, but the documentation says that I have to add an `Object` instance.

A: All `String` objects are `Object`s as well, so go ahead and add your string

Q: I know i added a `String`-object to my `ArrayList` but the `get()` method returns an `Object` object. What to do? I want to call `substring()` but there's no `substring()` method in `Object`

A: You have to use a type cast

```
String s = (String) list.get(index);
String s1 = s.substring(begin, end);
```

or even:

```
String s1 = ((String) list.get(index)).substring(begin,
end);
```

”Q&A” on Object

Q: I want to add a `String` object to my `ArrayList`, but the documentation says that I have to add an `Object` instance.

A: All `String` objects are `Object`s as well, so go ahead and add your string

Q: I know i added a `String`-object to my `ArrayList` but the `get()` method returns an `Object` object. What to do? I want to call `substring()` but there's no `substring()` method in `Object`

A: You have to use a type cast

```
String s = (String) list.get(index);
String s1 = s.substring(begin, end);
```

or even:

```
String s1 = ((String) list.get(index)).substring(begin,
end);
```

- Note: in Java 1.5: `new ArrayList<String>()` enables us to create list that may contains only `String` objects:

```
ArrayList<String> list = new ArrayList<String>();
```

Then `list` may also contain object from sunbclasses to `String` (that you must write yourselves . . .). `list.get()` will return objects of class `String`

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.
- A class can extend only one superclass but can implement any number of interfaces.

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.
- A class can extend only one superclass but can implement any number of interfaces.
- Each `ArrayList` is also a `List`. Basically all type compatibility rules are the same as for superclasses.

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.
- A class can extend only one superclass but can implement any number of interfaces.
- Each `ArrayList` is also a `List`. Basically all type compatibility rules are the same as for superclasses.
- You can think of interfaces as abstract classes with no member variables.

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.
- A class can extend only one superclass but can implement any number of interfaces.
- Each `ArrayList` is also a `List`. Basically all type compatibility rules are the same as for superclasses.
- You can think of interfaces as abstract classes with no member variables.
- The interface `java.util.List` extends the superinterface `java.util.Collection`, so every `ArrayList` is also a `Collection`

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.
- A class can extend only one superclass but can implement any number of interfaces.
- Each `ArrayList` is also a `List`. Basically all type compatibility rules are the same as for superclasses.
- You can think of interfaces as abstract classes with no member variables.
- The interface `java.util.List` extends the superinterface `java.util.Collection`, so every `ArrayList` is also a `Collection`
- Many methods declare generic interfaces as arguments:

Interfaces

- `ArrayList` *`implements`* `java.util.List` means that `ArrayList` 'promises' that it will implement at least the methods present in the interface `java.util.List`. A kind of contract that ensures that, among the methods available in `ArrayList`, we guarantee that those listed in `java.util.List` are present.
- A class can extend only one superclass but can implement any number of interfaces.
- Each `ArrayList` is also a `List`. Basically all type compatibility rules are the same as for superclasses.
- You can think of interfaces as abstract classes with no member variables.
- The interface `java.util.List` extends the superinterface `java.util.Collection`, so every `ArrayList` is also a `Collection`
- Many methods declare generic interfaces as arguments:

  `boolean addAll(Collection c)` is present in `ArrayList` and in all `Collections` adds all elements of the `Collection`. It doesn't care if the collection is a `List` or something else.

java.util.Iterator

- All `Collection` objects have an `iterator()` method, which returns an object from the class `java.util.Iterator`

java.util.Iterator

- All `Collection` objects have an `iterator()` method, which returns an object from the class `java.util.Iterator`
- The new `Iterator` positions itself ”before” the first element of the collection

java.util.Iterator

- All `Collection` objects have an `iterator()` method, which returns an object from the class `java.util.Iterator`
- The new `Iterator` positions itself ”before” the first element of the collection
- When `next()` is called it goes to the next element (the first time to the first object)

java.util.Iterator

- All `Collection` objects have an `iterator()` method, which returns an object from the class `java.util.Iterator`
- The new `Iterator` positions itself ”before” the first element of the collection
- When `next()` is called it goes to the next element (the first time to the first object)
- Before `next()`, it’s good to call `hasNext()`

java.util.Iterator

- All `Collection` objects have an `iterator()` method, which returns an object from the class `java.util.Iterator`
- The new `Iterator` positions itself ”before” the first element of the collection
- When `next()` is called it goes to the next element (the first time to the first object)
- Before `next()`, it's good to call `hasNext()`
- If there's no next element, a `NoSuchElementException` is thrown by `next()`

```
List lst = new ArrayList();
lst.add("first");
lst.add("second");
lst.add("third");
for (Iterator i = lst.iterator(); i.hasNext();)
   System.out.println(i.next());
```

java.util.Iterator

- All `Collection` objects have an `iterator()` method, which returns an object from the class `java.util.Iterator`
- The new `Iterator` positions itself ”before” the first element of the collection
- When `next()` is called it goes to the next element (the first time to the first object)
- Before `next()`, it's good to call `hasNext()`
- If there's no next element, a `NoSuchElementException` is thrown by `next()`

```
List lst = new ArrayList();
lst.add("first");
lst.add("second");
lst.add("third");
for (Iterator i = lst.iterator(); i.hasNext();)
   System.out.println(i.next());
```

- Some iterators also allow `remove()` of the current element

Mappings

- A List associates each object with an int index

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*
- Implementation of the `Map` interface: `java.util.HashMap`

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*
- Implementation of the `Map` interface: `java.util.HashMap`
  - `Object put(Object key, Object value)`
    associates a value with a given key.
    If another value was previously associated, it is returned, otherwise `null` is returned

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*
- Implementation of the `Map` interface: `java.util.HashMap`
  - `Object put(Object key, Object value)`
    associates a value with a given key.
    If another value was previously associated, it is returned, otherwise `null` is returned
  - `Object remove(Object key)` removes the element associated with the key, and returns it (or null if none is associated). clear() removes all.

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*
- Implementation of the `Map` interface: `java.util.HashMap`
  - `Object put(Object key, Object value)`
    associates a value with a given key.
    If another value was previously associated, it is returned, otherwise `null` is returned
  - `Object remove(Object key)` removes the element associated with the key, and returns it (or null if none is associated). clear() removes all.
  - /texttt`Object` get(Object key) returns the `Object` associated with the given key, or `null` if there is none

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*
- Implementation of the `Map` interface: `java.util.HashMap`
  - `Object put(Object key, Object value)`
    associates a value with a given key.
    If another value was previously associated, it is returned, otherwise `null` is returned
  - `Object remove(Object key)` removes the element associated with the key, and returns it (or null if none is associated). clear() removes all.
  - /texttt{Object get(Object key) returns the `Object` associated with the given key, or `null` if there is none
  - `int size()` returns the number of key-value pairs

Mappings

- A List associates each object with an int index
- A `java.util.Map` associates an object with any kind of Object called a *key*
- Implementation of the `Map` interface: `java.util.HashMap`
  - `Object put(Object key, Object value)`
    associates a value with a given key.
    If another value was previously associated, it is returned, otherwise `null` is returned
  - `Object remove(Object key)` removes the element associated with the key, and returns it (or null if none is associated). clear() removes all.
  - /texttt`Object` get(Object key) returns the `Object` associated with the given key, or `null` if there is none
  - `int size()` returns the number of key-value pairs
- `Map` also defines the interface `java.util.Map.MapEntry` managing key-value pairs

Mappings . . .

```
Map m = new HashMap();
m.put("k1", "v1");
m.put("k2", "v2");
for (Iterator i = m.entrySet().iterator(); i.hasNext();)
   Map.Entry me = (Map.Entry) i.next();
   System.out.println(me.getKey() + "->" + me.getValue());
```

Mappings . . .

```
Map m = new HashMap();
m.put("k1", "v1");
m.put("k2", "v2");
for (Iterator i = m.entrySet().iterator(); i.hasNext();)
   Map.Entry me = (Map.Entry) i.next();
   System.out.println(me.getKey() + "->" + me.getValue());
```

`java.util.Dictionary` is an obsolete version of `Map` (just to know what it is if you see one mentioned)

Mappings . . .

```
Map m = new HashMap();
m.put("k1", "v1");
m.put("k2", "v2");
for (Iterator i = m.entrySet().iterator(); i.hasNext();)
   Map.Entry me = (Map.Entry) i.next();
   System.out.println(me.getKey() + "->" + me.getValue());
```

`java.util.Dictionary` is an obsolete version of `Map` (just to know what it is if you see one mentioned)
Also `java.util.Enumeration` is an old version of Iterator. The concept is the same but `hasMoreElements()` instead of `hasNext()`, `nextElement()` instead of `next()` and no `remove()`!

Mappings . . .

```
Map m = new HashMap();
m.put("k1", "v1");
m.put("k2", "v2");
for (Iterator i = m.entrySet().iterator(); i.hasNext();)
   Map.Entry me = (Map.Entry) i.next();
   System.out.println(me.getKey() + "->" + me.getValue());
```

`java.util.Dictionary` is an obsolete version of `Map` (just to know what it is if you see one mentioned)
Also `java.util.Enumeration` is an old version of Iterator. The concept is the same but `hasMoreElements()` instead of `hasNext()`, `nextElement()` instead of `next()` and no `remove()`!
`java.util.Hashtable` is an older variant of `HashMap`

Mappings . . .

```
Map m = new HashMap();
m.put("k1", "v1");
m.put("k2", "v2");
for (Iterator i = m.entrySet().iterator(); i.hasNext();)
   Map.Entry me = (Map.Entry) i.next();
   System.out.println(me.getKey() + "->" + me.getValue());
```

`java.util.Dictionary` is an obsolete version of `Map` (just to know what it is if you see one mentioned)
Also `java.util.Enumeration` is an old version of Iterator. The concept is the same but `hasMoreElements()` instead of `hasNext()`, `nextElement()` instead of `next()` and no `remove()`!
`java.util.Hashtable` is an older variant of `HashMap`
`java.util.Properties`, a `Hashtable` subclass used to read and write configuration files (also in XML format in java 1.5 Properties!)

Threads: ”parallel” execution

Modern computer systems are essentially multi-user systems. Each user gets his/her own ”execution thread” and each such ”thread” gets a, in order, a small amount of time for each of its processes.

Threads: ”parallel” execution

Modern computer systems are essentially multi-user systems. Each user gets his/her own ”execution thread” and each such ”thread” gets a, in order, a small amount of time for each of its processes.
As Java supports this execution model it is denoted as ”multi threaded”. This means that a Java program that uses threads may run several threads simultaneously a thus give the impression of parallelism and if there are more than one processor actually run treads in parallel.

Threads: ”parallel” execution

Modern computer systems are essentially multi-user systems. Each user gets his/her own ”execution thread” and each such ”thread” gets a, in order, a small amount of time for each of its processes.
As Java supports this execution model it is denoted as ”multi threaded”. This means that a Java program that uses threads may run several threads simultaneously a thus give the impression of parallelism and if there are more than one processor actually run treads in parallel.
`java.lang.Thread` is built using the interface `java.lang.Runnable`

Threads: ”parallel” execution

Modern computer systems are essentially multi-user systems. Each user gets his/her own ”execution thread” and each such ”thread” gets a, in order, a small amount of time for each of its processes.
As Java supports this execution model it is denoted as ”multi threaded”. This means that a Java program that uses threads may run several threads simultaneously a thus give the impression of parallelism and if there are more than one processor actually run treads in parallel.
`java.lang.Thread` is built using the interface `java.lang.Runnable`
When a class is declared as an extension to the class `Thread` it gets two methods, one of which (`run()`) must be overridden. In `run()` you write the code that will be run in ”parallel”.

Threads: ”parallel” execution

Modern computer systems are essentially multi-user systems. Each user gets his/her own ”execution thread” and each such ”thread” gets a, in order, a small amount of time for each of its processes.
As Java supports this execution model it is denoted as ”multi threaded”. This means that a Java program that uses threads may run several threads simultaneously a thus give the impression of parallelism and if there are more than one processor actually run treads in parallel.
`java.lang.Thread` is built using the interface `java.lang.Runnable`
When a class is declared as an extension to the class `Thread` it gets two methods, one of which (`run()`) must be overridden. In `run()` you write the code that will be run in ”parallel”.
When you call the method `start()` you tell the system that you are ready and the system will run all `run()` methods will be called independently. Ypu may tell an object to ”sleep” a while by calling `static void sleep(long millis)` will halt the thread for `millis` milliseconds.

Parallel execution . . .

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }

    public void run() {
        // compute primes larger than minPrime
        . . .
    }
}
```

Parallel execution . . .

```
class PrimeThread extends Thread {
   long minPrime;
   PrimeThread(long minPrime) {
      this.minPrime = minPrime;
   }

   public void run() {
      // compute primes larger than minPrime
      . . .
   }
}
```

Start:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Parallel execution: alternate method

Let the class implement `Runnable`

```
class PrimeRun implements Runnable {
   long minPrime;
   PrimeRun(long minPrime) {
      this.minPrime = minPrime;
   }
   public void run() {
      // compute primes larger than minPrime
      . . .
   }
}
```

Parallel execution: alternate method

Let the class implement `Runnable`

```
class PrimeRun implements Runnable {
   long minPrime;
   PrimeRun(long minPrime) {
      this.minPrime = minPrime;
   }
   public void run() {
      // compute primes larger than minPrime
      . . .
   }
}
```

Start:

```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

That's all

You don't need to know all of Java. But

–

there is enough to learn anyhow