Summary Sun® Certified Programmer for Java 6 Study Guide - Katherine Sierra and Bert Bates

Chapter 1 – Declarations and Access Control 3
Chapter 2 – Object Orientation....9
Chapter 3 – Assignments....12
Chapter 4 – Operators 20
Chapter 5 – Flow control, Exceptions and Assertions....21
Chapter 6 – Strings I/O Formatting and Parsing 25
Chapter 7 – Generics and Collections....32
Chapter 8 – Inner Classes 42
Chapter 9 – Threads....45
Chapter 10 – Development 47

Chapter 1 – Declarations and Access Control

Identifiers

- Identifiers must start with a letter, a currency character($), or a connecting character (_). Identifiers can't start with a number
- After the first character, identifiers can contain any combination of letters, $'s, _'s or numbers
- No limit of the number of characters of an identifier
- You can't use a Java keyword as an identifier
- Identifiers are case sensitive

Java Keywords:

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

JavaBean

The three most important features of a JavaBean are the set of *properties* it exposes, the set of *methods* it allows other components to call, and the set of *events* it fires. Basically properties are named attributes associated with a bean that can be read or written by calling appropriate methods on the bean. The methods a Java Bean exports are just normal Java methods which can be called from other components or from a scripting environment. Events provide a way for one component to notify other components that something interesting has happened.

JavaBean Naming Conventions

1) If the property is not a boolean, the getter method's prefix must be *get*.
2) If the property is a boolean, the getter method's prefix is either *get* of *is*.
3) The setter method's prefix must be *set*.
4) To complete the name of the getter of setter, change the first letter of the property name to uppercase, and then append it to the appropriate prefix (get, is, or set)
5) Setter methods must be marked public, with a *void* return type and an argument that represents the property type
   Getter method signatures must be marked public, take no arguments, and have a return type that matches the argument type of the setter method for that property

JavaBean Listener Naming Rules

1) Listener method used to "register" a listener with an event source must use the prefix *add*, followed by the listener type. For example, *addActionListener(ActionListener l)* is a valid name for a method that an event source will have to follow others to register for *ActionEvents*
   XxxEvent addXxxListener
2) Listener method names used to *remove* ("unregister") a listener must use the prefix remove, followed by the listener type. *removeActionListener(ActionListener l)*

XxxEvent removeXxxListener

3) The type of listener to be added of removed must be passed as the argument to the method.

Source file declaration rules

1) There can be only one public class per source code file
2) Comments can appear at the beginning or end of any line in the source code file
3) If there is a public class in a file, the name of the file must match the name of the public class.
4) Package statement must be the first statement in the source code file.
5) Import statement between the *package statement* and the *class declaration*
6) Import & Package statement apply to all classes in the file
7) A file can have more than one nonpublic class
8) Files with no public classes can have a name that does not match any of the classes in the file

Member variable, Instance variable, Property

A Class can have different members
1) a member variable
2) a static member variable
3) a member method
4) a static member method
5) an inner class

Note: a *class variable* is often called a *static member variable*

An Object (instance of a certain Class) can have
1) an instance variable
2) an instance method
3) an inner class

Note: a *member variable* is often called an *instance variable* and vice versa

A JavaBean has the following items:
1) a property
2) a method
3) an event

Note: a *property* is often called a *member variable* or an *instance variable*

Class Modifiers

Acces modifiers:

1) public
2) protected (only *Inner Classes*)
3) private (only *Inner Classes*)

4 levels of control: *default* is not an explicit modifier

Non-access modifiers:
1) strictfp
2) final
3) abstract

Interface:
- All methods are by default *public abstract* although it doesn't have be mentioned

- All variables are *public*, *static* and *final* by default
- Because interface methods are abstract, they cannot be marked *final*, *strictfp* or *native*
- An interface can extend one or more other interfaces
- An interface can extend extend anything but another interface
- An interface cannot implement another interface or class
- An interface must be declared with the keyword interface
- Interface types can be used polymorphically
- A constant is declared by *public static final* but the keywords don't have to appear in the interface-file

Access to Class Members:

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the package	Yes	Yes, through inheritance	No	No
From any non-subclass outside the package	Yes	No	No	No

example:

```
--------------------------
package foo;
public class Parent {
      protected String pa = "vader";
      protected String getPa() {
            return pa;
      }
      protected void setPa(String pa) {
            this.pa = pa;
      }
}
---------------------------
package foo;
public class Child extends Parent {}
---------------------------
package baa;
import foo.Child;

public class SmallChild extends Child {
   public static void main(String... args){
      Child kind = new SmallChild();
      System.out.println(kind.pa); // WON'T COMPILE

      // pa is protected, so it can be used from a subclass of another
      // package, but it cannot be used via a reference of a parent class.

      SmallChild kindje = new SmallChild();
      System.out.println(kindje.pa); // WILL COMPILE
   }
}
---------------------------
```

Nonaccess Member Modifiers:

final
- final methods: cannot be overridden in a subclass
- final arguments of methods: cannot be changed (reassigned) a new value inside the method
- final class: cannot be sub-classed
- final member variable: cannot be reassigned a new value and has to be initialized when an instance is created and before the constructor completes.

abstract
- abstract method is a method that has been *declared* but not *implemented*
- In there is one *abstract method* then the class has to be declared *abstract*
- The first concrete subclass must implement *all abstract methods* of the super-class
- An abstract method ends in a semicolon instead of curly braces
- If an abstract class *extends* another abstract class it doesn't have to define or implement the abstract methods
- A combination of abstract and final is not possible (although an *abstract class* can have a *final non-abstract method)*
- A combination of abstract and private is not possible -> abstract means it has to be overridden, private means it is not possible to override because it is not visible

synchronized
- The method can only be accessed by one thread at a time
- It can be combined with the 4 access modifiers (public, default, private, protected)

native
- the method is implemented in a platform dependent code (often C)
- the body must contain a semicolon (= not implemented)

strictfp
- forces floating points to adhere to the IEEE 754 standard
- only for classes and methods, not variables

static
- will create a class variable or a class method that is independent of any instances created for the class
- a static instance variable is not possible: because it will be of the class not of the instance

transient Variable
- It is skipped by the JVM when serializing the object

volatile Variable
- Tells the JVM that the thread accessing the variable has to reconcile its own copy of the variable with the master copy in memory

Variable Argument Lists
- It must be the last parameter in an argument signature
- For example public void doStuff(int a , int… b)
- There can only be one variable argument in a methods signature

Variable Declarations
1) char
2) boolean
3) byte
4) short
5) int
6) long

7) double
8) float

Ranges of numeric Primitives

Type	Bits	Bytes	Minimum range	Maximum range
byte	8	1	-2^7	2^7 - 1
short	16	2	-2^15	2^15 – 1
int	32	4	-2^31	2^31 – 1
long	64	8	-2^63	2^63 – 1
float	32	4	n/a	n/a
double	64	8	n/a	n/a
char	16	2	0	2^16

The following modifiers (11 in total) are allowed

	final	public	protected	private	static	transient	volatile	abstract	synchronized	strictfp	native
Classes (4)	final	public						abstract		strictfp	
Constructors (3)		public	protected	private							
Enums (1)		public									
Constructors enum (1)				private							
Local Variables (1)	final										
Variables (non local) (7)	final	public	protected	private	static	transient	volatile				
Methods (9)	final	public	protected	private	static			abstract	synchronized	strictfp	native
Inner Classes (7)	final	public	protected	private	static			abstract		strictfp	
Method Local Inner Classes (2)	final							abstract			

Member variables cannot be: abstract, native, synchronized or strictfp
Member methods cannot be: transient, volatile

Declaring an Array

- int[] key
- int key[]
- never include a size when declaring an Array

Static Variables and Methods

Possible	Not Possible
Methods	Constructor
Variables	Classes
A class nested within another class	Interfaces
Initialization blocks	Method local inner classes
	Inner class methods and instance variables
	Local variables

Enums

- Enums can have instance variables, methods and constructors
- An enum has to start with the declaration of values
- A constructor of an enum cannot access a non-final static field
- The compiler doesn't add a no-argument constructor, if there is another constructor
- An enum has a *values()* method returning an array with values
- An enum has an *ordinal()* method returning its position in the enum declaration.
- An enum has an *valueOf* methode to convert a String to the corresponding enum value.
- Enums can have constructors but can never be invoked directly
- Enum constructor can only be *private* or *default*
- The constructor can have more than one argument
- Enum constructors can be overloaded
- Look out for the semicolon " ;" if after the constants there is more code (variables, methods, constructors)
- An enum value cannot be a string (e.g. enum Bla {"a", "b"} is not possible)
- The values of an enum can be considered as constants (public final static).
- A value of an enum is not a *String* or an *int* (see example: BIG is of type *CoffeeSize*)

Enum can be declared outside a class

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };

class Coffee {

   CoffeeSize size;

}

public class CoffeeTest1 {

     public static main void (String args[]) {

        Coffee drink = new Coffee();

        drink.size = CoffeeSize.BIG;
     }

}
```

Enum can be declared inside a class

```
class Coffee2 {

   enum CoffeeSize { BIG, HUGE, OVERWHELMING };

   CoffeeSize size;

}


public class CoffeeTest2 {

   public static main void (String args[]) {

      Coffee drink = new Coffee2();

      drink.size = Coffee2.CoffeeSize.BIG;
   }

}
```

Enum can be declared in it's own file (CoffeeSize.java):

```
public enum CoffeeSize { BIG, HUGE, OVERWHELMING };
```

Chapter 2 – Object Orientation

Reference Variables

- A reference variable can be of only one type, and once declared, can never be changed
- A reference is a variable, so it can be reassigned to different objects (unless declared final)
- A reference variable's type determines the methods that can be invoked on the object the variable is referencing (this is known at compile time)
- A reference variable can refer to any object of the same type as the declared reference, or it can refer to a subtype of the declared type (passing the IS-A test)
- A reference variable can be declared as a class type or as an interface type. If the reference variable is declared as an interface type, it can reference any object of any class that *implements* the interface (passing the IS-A test)

Rules for overriding a method

- The overridden method has the same name.
- The argument list must exactly match (i.e. *int, long* is not the same as *long, int*) that of the overridden method. If the don't match, you end up with an overloaded method.
- The order of arguments is important
- The return type must be the same as, or a subtype of, the return type declared in the original overridden method in the super-class.
- The access level can't be more restrictive than the overridden method's
- The access level CAN be less restrictive than that of the overridden method
- Instance methods can be overridden only if they are inherited by the subclass. A subclass within the same package as the instance's super-class can override any super-class method that is not marked private or final. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass)
- Trying to override a private method is not possible because the method is not visible, that means that a *subclass can define a method with the same signature* without a compiler error!
- Trying to override a final method will give a compile error
- The overriding method CAN throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception
- The overriding method must NOT throw checked exceptions that are new or broader than those declared by the overridden method
- The overriding method can throw narrower or fewer exceptions.
- You cannot override a method marked final
- You cannot override a method marked static
- If a method is not visible it cannot be inherited.
- If a method cannot be inherited it cannot be overridden.
- An overriding method CAN be final

Overloaded methods

- Overloaded methods have the same name
- Overloaded methods must change the argument list
- Overloaded methods can change the return type
- Overloaded methods can change the access modifier
- Overloaded methods can declare new or broader checked exceptions

Which method is called

- Which overridden version of the method to call is decided at runtime based on the object type.
- Which overloaded version of the method to call is based on the reference type of the argument passed at compile time

Implementing an Interface

- Provide concrete implementations for all the methods from the declared interface
- Follow all the rules for legal overrides
- Declare no checked exceptions on implementation methods other than those declared by the interface method, or subclasses of those declared by the interface method.
- Maintain the signature of the interface method, and maintain the same return type (or a subtype).
- If the implementing class is abstract, then the methods don't have to appear in that class but in the first concrete class in the inheritance tree

Reference Variable Casting

- Downcasting: casting down the inheritance tree (explicitly declare the type)
- Upcasting: casting up the inheritance tree (implicitly: you don't have to type in the cast)

Overridding and Covariant Returns

- You can override a method and change the return type as long as the return type is a subclass of the one declared in the overridden method

Returning a method's value

1) You can return null in method with an object reference return type
2) An array is a legal return type
3) In a method with a primitive return type, you can return any value or variable that can be implicitly converted to the declared type
4) In a method with a primitive return type, you can return any value or variable that can be explicitly cast to the declared return type

```
public short getInt(){
    int s = 5;
    return s;    // doesn't compile, needs a cast
}

public short getInt(){
   return 5;    // does compile it can explicitly cast to the declared
                // return type
}
```

5) You must not return anything from a method with a void return type
6) In a method with an object reference return type, you can return any object type that can be implicitly cast to the declared return type

Constructors

1) Constructors can use any access modifier, including private
2) The constructor name must match the name of the class

3) Constructors must not have a return type
4) It's legal to have a method with the same name as the class
5) If you don't type a constructor into you class code, a default constructor will be automatically be generated by the compiler
6) The default constructor is always a no-argument constructor
7) If you want a no-argument constructor and you have typed any other constructor(s) into your class code, the compiler won't provide the no-argument constructor
8) Every constructor has, as its first statement, either a call to an overloaded constructor (this()) or a call to the super-class constructor (super())
9) If you create a constructor, and you do not have an explicit call to super() or an explicit call to this(), the compiler will insert a no-argument call to super(). (if there is no no-argument constructor in the super-class, a compile error will be generated).
10) A call to super() can be either a no-argument call or can include arguments passed to the super constructor
11) A no-argument constructor is not necessarily the default constructor, although the default constructor is always a no-argument constructor
12) You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs
13) Only static variables and methods can be accessed as part of the call to super() or this().
14) Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated
15) Interfaces do not have constructors.
16) The only way a constructor can be invoked is from within another constructor.

Coupling and Cohesion

- Coupling: is the degree that one class knows about another (loose coupling is better, use the API)
- Cohesion: used to indicate the degree to which a class has a single, well focused purpose (high cohesion is better, easier to maintain: less frequently changed)

Chapter 3 – Assignments

Primitive assignments

- Octal literals begin with a '0': example: int nine = 011; (decimal 9)
- Hexadecimal literals begin with a '0X' or '0x' example: int fteen = 0xf; (decimal 15)
- Floating point by default doubles, if float: example: float f = 34.45544**F;**
- Chars, Unicode (16 bits) example: char N = '\u004E';
- Chars 0 to 65000, compile error without cast example: char c = (char) 70000;

Casting

Implicit cast
When you are widening a conversion: from a byte to an int

Explicit cast
When you are narrowing a conversion: from a double to a float

- Literal integer (e.g. 7) is implicitly a int, cast is done by the compiler
example: char d = 27;

- Adding two bytes can't be assigned to a byte without a cast. The result of a calculation with operands of type smaller then int will be promoted to an int, that is why the cast is necessary.
byte a = 10;
byte b = 2;
byte c = **(byte)** (a + b); // you have to put the explicit cast

c+=6; // This is possible without a cast
c=120; // Although 120 is an implicit int, you don't need a cast

Scope of variables

1) Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the JVM
2) Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed
3) Local variables are the next; they live as long as the method remains on the stack
4) Block variables live only as long as the code block is executing

Most common scoping errors

1) Attempting to access a instance variable from a static context (typically main())
2) Attempting to access a local variable from a nested method
3) Attempting to access a block variable after the code block has completed

Default values for Primitive types and Reference types and Static variables

Variable Type	Default Value
Object reference	null
byte, short, int, long	0
float, double	0.0
boolean	false
char	'\u0000' is value 0 (char is unsigned number)

Array elements are always initialized with default values like instance variables
Static member variables are also initialized with default values like instance variables

Assignments and Strings

- String objects are immutable
- When you modify a String the following will happen:
    1. A new String is created (or a matching String is found in the String pool) leaving the original String untouched
    2. The reference used to modify the String is then assigned to a new String object

Pass-By-Value

The called method can't change the caller's variable (it gets its own copy)

1. when the variable is a primitive, the method gets its local copy and it can't change the original variable (primitive)
2. when the variable is a reference, the method can't reassign the original reference variable (although it can change the contents of the object referred to)

Arrays

Declaring:
1) int[] key;
2) int key [];

Constructing (need to specify a size)
1) int[] key = new int[4];
2) int [] [] myList = new int[4][] (-> only the first one must be assigned a size)

Initializing
1) An array with primitives: its elements are always with default values (0, 0.0, false, '\u0000')
2) Declaring constructing and initializing at once: int[] [] myList = {{5,2,4,7}, {9,2}, {3,4}};

Constructing and Initializing an Anonymous Array
1) int [] testscores;
   testscores = new int[] {2,4,7};

Init Blocks

1) Init blocks execute in the order they appear
2) Static init blocks run once, when the class is first loaded
3) Instance init blocks run everytime a class instance is created
4) Instance init blocks run after the constructor's call to super and before the body of the

constructors code:

example:

```
public class Parent {
	static {
		System.out.println("Staticblock Parent ");
	}
	{ System.out.println("Initblock Parent ");	}
	public Parent() {
		System.out.println("Constructor Parent ");
	}
	static {
		System.out.println("Staticblock Parent 2");
	}
}
class Child extends Parent {
	static {
		System.out.println("Staticblock Child ");
	}
	{ System.out.println("Initblock Child "); }
	public Child() {
		this("A");
		System.out.println("Constructor Child ");
	}
	public Child(String a){
		System.out.println("Constructor Child " + a);
	}
	public static void main(String args[]) {
		new Child();
	}
	{ System.out.println("Initblock Child 2"); }
}
```

Output:

```
Staticblock Parent
Staticblock Parent 2
Staticblock Child
Initblock Parent
Constructor Parent
Initblock Child
Initblock Child 2
Constructor Child A
Constructor Child
```

Wrapper Classes

Primitive	Bits	Wrapper Class	Constructor Arguments
boolean	Undefined	Boolean	boolean or String
byte	8	Byte	byte or String
short	16	Short	short or String
char	16 (unsigned)	Character	char
int	32	Integer	int or String
float	32	Float	float, **double** or String
long	64	Long	long or String
double	64	Double	double or String

Wrapper Methods

xxxValue:	To convert a value of a wrapped numeric to a primitive
parseXxx(String s):	Takes a string (optional radix, e.g. 2, 10, 16) and returns a primitive (throws NumberFormatException)
-valueOf()	Takes a primitive or a string (optional radix, e.g. 2, 10, 16) and returns a wrapper object (throws NumberFormatException)

Integer, Byte, Short and Long Wrapper classes
-valueOf(primitive p)
-valueOf(String s)
-valueOf(String s, radix r) e.g. `Integer octal = Integer.valueOf("20", 8);`

Float and Double Wrapper classes
-valueOf(primitive p) e.g. `double d = 1.1; Double big = Double.valueOf(d);`
-valueOf(String s) e.g. `Double t = Double.valueOf("1.1");`

Character Wrapper class
-valueOf(primitive p) e.g. `char a= 'a'; Character aa = Character.valueOf(a);`

String toString(): Returns the string representation of the value in the wrapped object
static String toString(prim p) e.g. `Double.toString(3.14);`
static String toString(prim p, radix) e.g. `Long.toString(214, 2);`

Integer and Long Wrapper classes
String toHexString(int i) e.g. `Integer.toHexString(254);`
String toBinaryString(int i) e.g. `Integer.toBinaryString(254);`
String toOctalString(int i) e.g. `Integer.toOctalString(254);`

Wrapper and Equals

To save memory two instances of the following wrapper objects will always be == when their primitive values are the same:

```
Integer i3= 10;
Integer i4= 10;
if (i3==i4) {
   System.out.println("Same");
}
```

This will print Same

Watch out: this is not the case if you create the objects yourself

```
Integer i3=new Integer(10);
Integer i4=new Integer(10);
if (i3==i4) {
   System.out.println("Same");
}
```

This won't print anything

This is the same for the following wrappers:
1) Boolean
2) Byte
3) Char (values from '\u0000' to '/u007f'
4) Short (values from -128 to 127)
5) Integer (values from -128 to 127)

Autoboxing

Autoboxing: the java compiler automatically does boxing and unboxing where necessary

Boxing: wraps a primitive to a corresponding Wrapper object
example: Integer i = 10; (before java 5: Integer i = new Integer(10);)

Unboxing: unwraps a Wrapper object to a primitive
example:

```
Boolean bool = true;
if (bool) {
   System.out.println("unboxing in for loop");
}
```

Widening

The JVM tries to make a match, but if the match can't be made it looks for the method with the smallest argument that is wider than the parameter.

example:

```
public void increase (int i){}
public void increase (long i){}

public static void main (String args[]){
      byte b= 5;
      increase(b); // will use increase(int)
}
```

Overloading with boxing and var args

Widening has priority over boxing
example:

```
public void increase (Integer i){}
public void increase (long i){}

public static void main (String args[]){
      int b= 5;
      increase(b); // will use increase(long)
}
```

Widening has priority over variable arguments
example:

```
public void increase (int i, int j){}
public void increase (int… i){}

public static void main (String args[]){
      byte b= 5;
      byte c= 6;
      increase(b,c); // will use increase(int, int)
}
```

Boxing has priority over variable arguments
example:

```
public void increase (Byte i, Byte j){}
public void increase (Byte… i){}
```

```
public static void main (String args[]){
      byte b= 5;
      byte c= 6;
      increase(b,c); // will use increase(Byte, Byte)
}
```

Wrappers can not be widened
example:

```
public void increase (Long i){}

public static void main (String args[]){
      Integer b= 5;
      increase(b); // IS NOT LEGAL
}
```

Widening an Boxing is not allowed
example:

```
public void increase (Long i){}

public static void main (String args[]){
      int b= 5;
      increase(b); // IS NOT LEGAL
}
```

Boxing and Widening is allowed

```
static void go (Object o) {
   Byte b2 = (Byte) o;
   System.out.println(b2);
}
public static void main (String args[]){
      byte b= 5;
      go(b);  // boxing to Byte ( Byte is-a Number is-a Object )
}
```

Widening with Variable Arguments is allowed

```
static void wide_varargs (long... x) {
   System.out.println("long... x");
}

public static void main (String args[]){
      int b= 5;
      wide_varargs (b, b); // will print long... x
}
```

Boxing with Variable Arguments is allowed

```
static void boxing_var (Integer... x) {
   System.out.println("Integer... x");
}

public static void main (String args[]){
      int b= 5;
      boxing_var (b, b);  // will print Integer... x
}
```

Primitives and Variable Arguments can be ambigious

```
static void overload(int... d) {
   System.out.println("Integer");
}
```

```
static void overload(long... d) {
   System.out.println("Long");
}

public static void main (String args[]){
      int i = 1;
      overload(i); // DOES NOT COMPILE
}
```

Rules widening and boxing

1) Primitive widening uses the smallest method argument possible
2) Used individually, boxing and var-args are compatible with overloading
3) You cannot widen from one wrapper type to another (IS-A fails)
4) You cannot widen and then box
5) You can box and widen
6) You can combine var-args with either widening or boxing

Garbage collector

1) You can request the JVM to execute the garbage collector, but you will never know whether it will do so. (request is made by calling *System.gc();* or *Runtime.getRuntime().gc();*)
2) Strings are never gc-ed as they are in a String-pool

Object.finalize()

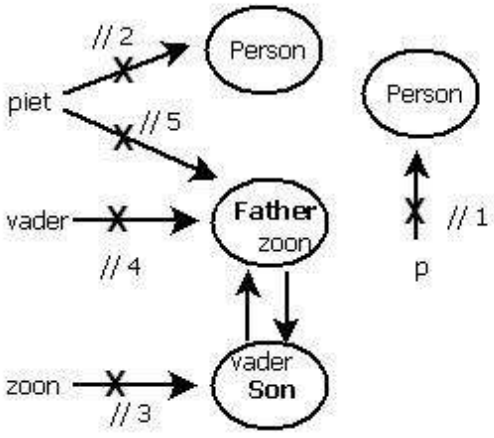
- For any given object finalize() will be called only once (at most) by the garbage collector
- Calling finalize() can actually result in saving an object from deletion
- There is no guarantee that the method will ever run
- Exceptions during execution of the method are swallowed

Garbage Collection

An object is eligible for garbage collection when no live thread can access it.
This happens when

- a reference variable that refers to the object is set to null.
- a reference variable is reassigned to another object.
- objects created in a method (and *not* returned by the method) after the method finishes
- islands of isolation, there is no reference to the island of objects

```
public class Father extends Person {
      Son zoon;
      public void uselessMethod() {
            Person p = new Person();
      }
      public static void main(String args[]) {
            Father vader = new Father();
            vader.uselessMethod();          // 1
            Son zoon = new Son();
            Person piet = new Person();
            vader.zoon = zoon;
            zoon.vader = vader;
            piet = vader;          // 2
            zoon = null;             // 3
            vader = null;               // 4
            piet = null;                  // 5
      }
}
class Son extends Person {Father vader;}
class Person {}
```



After line // 5 the island (Father and Son) is eligable for GC

Chapter 4 – Operators

- There are six relational operators: <, <=, >, >=, !=, ==
- The *instanceof* operator is for object references, for interfaces if any of the superclasses implements the interface
- The | of & always evaluate both operands
- The ^ is the exclusive or: only true if exactly one of the operands evaluate true
- String Concatenator: if one of the operands is String it will concatenate the operands
  System.out.println(4 + 5 + " "); // prints 9, + is left associative
  System.out.println(4 + " " + 5); // prints 45

Chapter 5 – Flow control, Exceptions and Assertions

Flow Control

if
if (booleanExpression) statement1
else if (booleanExpression) statement2
..
else statement3

- Look out for an assignment in an if condition:
  example:
```
boolean b = false;
if (b = true) { System.out.println("ja"); }
else {System.out.println("nee");} // prints ja
```

switch

```
switch (expression) {
  case constant1: code block
  case constant2: code block
  default: code block
}
```

- use break to not evaluate next constants
- a switch expression must evaluate to a ***char, byte, short, int, enum***
- default doesn't have to be the last switch statement
- a case constant has to be a compile time constant
  example:
```
final int a = 1;
final int b;
b = 2;
int x = 0;
switch (x) {
   case a: {} // ok
   case b: {} // compile error;
}
```

Loops

Code in Loop	What Happens
break	Execution jumps immediately to the first statement after the for loop
return	Execution jumps immediately to the calling method
System.exit()	All program execution stops, JVM shuts down

for

```
for (/* initialization */ ; /* Condition */ ; /* Iteration * /) {
  // loop body
}
```

- none of the three sections is required: for (; ;) { // is allowed}
- watch out for scoping issues:
  example:
```
for (int i = 0; i < 5 ;i++) System.out.println("ja");
```

```
System.out.println(i + "not allowed");  // i is out of scope!!
```

example2:

```
int i;
for (i = 0; i < 5 ;i++) System.out.println("ja");
System.out.println(i + "allowed");  // i is in scope!!
```

- Enhanced loop is for arrays and collections:
  for (declaration ; expression)
  example:

```
void playSongs(Collection<Song> songs) {
    for ( Iterator< Song > i = songs.iterator(); i.hasNext(); )
        i.next().play();
    }
```

  can be rewritten like this:

```
void playSongs(Collection<Song> songs) {
    for ( Song s:songs )
        s.play();
    }
```

- break: stops the entire loop
- continue: go to the next iteration
- Labelled statements: continue and break statements must be inside a loop that has the same label name; otherwise the code won't compile

Exceptions

```
try {
   // do stuff
} catch (someException) {
  // do exception handling
} finally {
  // do clean up
}
```

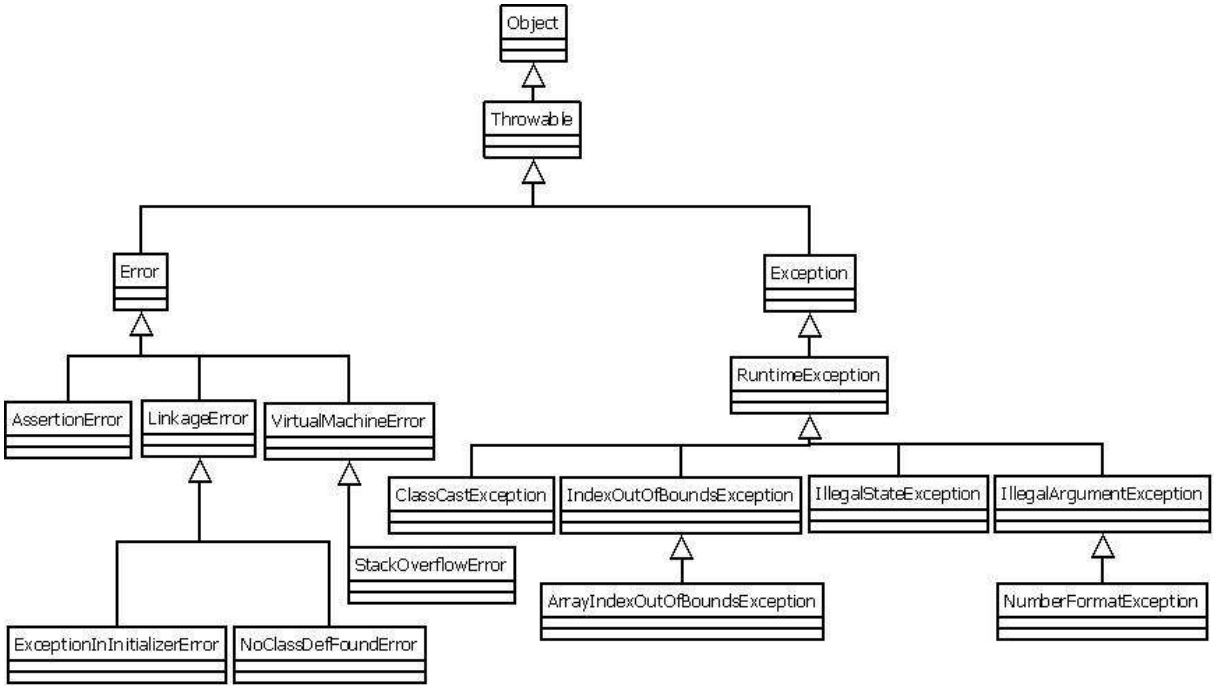
- A try without a catch or a finally is not allowed
- code in between try and catch is not allowed
- in the catch block a specific exception has to come before a general (supertype) exception (otherwise compile error)
- Any method that might throw an exception (unless it is a runtime exception) has to be declared
- All *Non Runtime Exceptions* are checked exceptions
- Each method must handle either all checked exceptions by supplying a catch clause or list each unhandled checked exception as a thrown exception (Handle or Declare Rule)

Exceptions come from

1) JVM exceptions – exceptions or error thrown by the JVM
2) Programmatic exceptions – thrown explicitely by application or API programmers

Exception	Description	Typically thrown by
ArrayIndexOutOfBoundsException	invalid index of an array	JVM
ClassCastException	invalid cast of reference variable to a type that doesn't pass the IS-A test	JVM
NullPointerException	invalid acces of an object via a reference who's value is null	JVM
ExceptionInInitializerError	invalid initialization in init block or static variable	JVM

StackOverflowError	method recurses to deeply	JVM
NoClassDefFoundError	JVM can't find a .class file	JVM
IllegalArgumentException	method gets an argument formatted differently then the method expects	Programmatically
IllegalStateException	state of the environment doesn't match the operation being attempted. e.g. Scanner that has been closed	Programmatically
NumberFormatException	thrown when a string is not convertable to a number	Programmatically
AssertionError	thrown when the statement's boolean test returns false	Programmatically



Exceptions and Errors

Assertion

- Always assert that something is true
- Assertions are disabled by default
- example 1:
  - assert (y>x)
  - … // code assuming y>x
- example 2:
  - assert (y>x): "y is: " + y + "x is: " + x; // expression String: ends with a semi-colon
  - … // code assuming y>x // anything that returns a value is allowed

Assert is in Java1.3 an Identifier and in Java1.4 and Java5 a Keyword

- if you use assert as an Identifier then you have to tell the compiler:
  - javac –source 1.3 bla/TestJava.class -> it will issue warnings that assert is used as a keyword
  - javac –source 1.4 bla/TestJava.class -> it will issue errors (assert is a keyword)
  - javac bla/TestJava.class -> it will issue errors (assert is a keyword)

    - javac –source 1.5 bla/TestJava.class -> it will issue errors (assert is a keyword)
    - javac –source 5 bla/TestJava.class -> it will issue errors (assert is a keyword)
- if you use assert as an Keyword in java 1.3
    - javac –source 1.3 bla/TestJava.class -> it will issue errors (keyword doesn't exist in 1.3)

Enabling assertions

1) java –ea bla/TestJava.class
2) java –enableassertions bla/TestJava.class
3) selective enabling, disabling:
    - java –ea –da:bla/blie // assertions, but not for bla/blie
    - java –ea –dsa // assertions, but not for system classes
    - java –ea –da:bla/blie… // assertions, but disable bla/blie and subpackages

Appropriately use of assertions

- Don't use assertions to validate arguments to a public method
- Do use assertions to validate arguments to a private method
- Don't use assertions to validate command-line arguments
- Do use assertions even in public methods, to check for cases that you know are never, ever suppose to happen

Chapter 6 – Strings I/O Formatting and Parsing

Strings

- Are immutable e.g. String x = “abcdef”; x = x.concat(“g”); will create a new String “abcdefg” and the reference to the original Strings are lost.
- String s = new String(“abc”); will create two objects: a String object in (non-pool) memory and a literal in the pool-memory

Methods on Strings

charAt()	Returns the character on the specified index
concat()	Appends one string to another (just like “+”)
equalsIgnoreCase()	determines the equality of two strings (ignoring the case)
length()	returns the number of characters of the string
replace()	replaces occurences of a character with a new character
substring()	Returns part of a string
toLowerCase()	Returns a string with uppercase characters converted
toString()	Returns the value of a string
toUpperCase()	Returns a string with lowercase characters converted
trim()	Removes whitespace from the ends of the string

substring(int a, int b) a – starting index (zero based), b – ending index (non zero based)
example:

```
public static void main(String args[]) {
      String string = "substring";
               // index 0(s) 1(u) 2(b) 3(s) 4(t) 5(r) 6(i) 7(n) 8(g)
      System.out.println(string.substring(1,4); // prints ”ubs”
}
```

Arrays have an attribute length, not a method length()

String x = “test”;
System.out.println(x.length); // compile error

String[] x = new String[3];
System.out.println(x.length()); // compile error

StringBuffer and StringBuilder

- Use the classes when there is a lot of string manipulations (e.g. File I/O)
- Stringbuffer’s methods are thread safe
- Same API
- substring(a,b) returns a string so it cannot be used inside a chain

Methods:

public synchronized StringBuffer append(String s)	will update the value of the object (takes also other types like int, float…)
public StringBuilder delete(int start, int end)	will remove substring from start to end -> both zero-based
public StringBuilder insert(int offset, String s)	insert string in object at offset (zero-based)
public synchronized StringBuffer reverse()	reverses the value of the StringBuffer object

public String toString() returns the value of the StringBuffer object

File Navigation and I/O

File:	Abstract representation of file and directory names
FileReader:	This class is used to read character files
BufferedReader:	Read large chunks of data from a file and keep this data in a buffer (minimizing I/O)
FileWriter:	This class is used to write characters or Strings to a file
BufferedWriter:	Write large chunks of data to a file (minimizing I/O)
PrintWriter:	An enhanced writer (no need of File being wrapped into BufferedWriter/FileWriter. Constructor takes a File or a String.

File (use in a try catch block with IOException)

public boolean createNewFile()	this method creates a new file if it doesn't already exists
public boolean exists()	checks if the file exists
public boolean delete()	deletes a file or directory (if empty)
public boolean isDirectory()	checks if the file is a directory
public boolean isFile()	checks if the file is a file
public String[] list()	lists the files in a directory, if the File is not a dir it returns null
public File[] listFiles()	same as list expect returns a File[] instead of String[]
public boolean mkdir()	creates a directory from the abstract pathname
public renameTo(File f)	renames a file or directory (even if not empty)
public boolean mkdirs()	creates directories including non existent parent dirs

Two ways of creating a file

1) call createNewFile() on a File object
2) create a FileReader or FileWriter or PrintWriter of FileInputStream or FileOutputStream

FileWriter and FileReader

example:

```
try {
      File f = new File("name");
      FileWriter fw = new FileWriter(f);
      fw.write("These are \n a few \n rules");
      fw.flush(); // flush before closing
      fw.close(); // you have to close the FileWriter
 } catch  (IOException e) { }
```

Java io.class	Key Constructor(s) Arguments
File	File, String // parent, child String String, String // parent, child
FileWriter	File, String
BufferedWriter	Writer
PrintWriter	File // as of Java 5 String // as of Java 5 OutputStream Writer
FileReader	File String
BufferedReader	Reader

Console

In Java 6 there is a new object called the java.io.Console.

- It provides methods to access the character-based console device, if any, associated with the current Java Virtual Machine (JVM).
- The console is not always there, it depends upon the underlying platform and the manner in which the JVM is invoked:
  - if the JVM is started from an interactive command line then its console will exist (if it doesn't redirect the standard input and output streams)
  - if the JVM is started automatically, for example by a background job scheduler, then it will typically not have a console.
- At his moment Eclipse (version 3.5) is returning *null* when requesting the Console object.
- The readPassword() method returns a char[] so that you can easily remove it from memory. A String might still live on in the "pool".

java.io.Console	Description
public String readLine()	Reads a single line of text from the console.
public String readLine(String fmt, Object... args)	Provides a formatted prompt, then reads a single line of text from the console.
public char[] readPassword(String fmt, Object... args)	Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
public char[] readPassword(String fmt, Object... args)	Provides a formatted prompt, then reads a password or passphrase from the console with echoing disabled.
public Console format(String fmt, Object... args)	Writes a formatted string to this console's output stream using the specified format string and arguments.
public Console printf(String format, Object... args)	A convenience method to write a formatted string to this console's output stream using the specified format string and arguments.
public PrintWriter writer()	Retrieves the unique PrintWriter object associated with this console.
public Reader reader()	Retrieves the unique Reader object associated with this console.
public void flush()	Flushes the console and forces any buffered output to be written immediately.

Serialization

1) Transient instance variables are never serialized
2) Use the two basic methods to serialize/deserialize
3) When you are a serializable class but your superclass isn't, then any instance variables you inherit from that superclass will be reset to the values that were given during the original construction, because the superclass constructor will run!

Two basic methods:
1) ObjectOutputStream.writeObject() - Serialize and write
2) ObjectInputStream.readObject() - Read and deserialize

example:

```
import java.io.*

class Cat implements Serializable {}
```

```
public class SerializeCat {

      public static main void (String[] args) {
            Cat c = new Cat();
         try {
            FileOutputStream fo = new FileOutputStream("testSer");
            ObjectOutputStream oo = new ObjectOutputStream(fo);
            oo.writeObject(c);
            oo.flush();
            oo.close();
         } catch (IOException) {}
         try {
            FileInputStream fi = new FileInputStream("testSer");
            ObjectInputStream oi = new ObjectInputStream(fi);
            Cat d = (Cat) oi.readObject();
            oi.close();
         } catch (IOException) {}
}
```

To add extra functionality to the (default) serialization method, use the following methods:
private void writeObject (ObjectOutputStream oo)
private void readObject (ObjectInputStream oi)
Don't close the ObjectOutputStream in those methods!

Dates, Numbers and Currency

The following classes are important:

1) java.util.Date	An instance of Date represents a mutable date and time to a milisecond. Mostly used to bridge between a Calendar and DateFormat
2) java.util.Calendar	This class has a variety of methods to convert and manipulate dates and times
3) java.text.DateFormat	This class is used to format dates to various locales in the world
4) java.text.NumberFormat	This class is used to format numbers and currencies for various locales in the world
5) java.util.Locale	This class is used in conjunction with DateFormat and NumberFormat to format dates, numbers and currency for specific locales.

Date
Default constructor and a constructor with a long (number of miliseconds since 1970)

Calendar
No constructor, but a factorymethod
c.getInstance()
c.getInstance(Locale l)

c.set(int year, int month, int day)	month is zero-based
c.add(Calendar.MONTH, 4)	add a month to the date
c.roll(Calendar.MONTH, 9)	add 9 months to the date without affecting the year
Date c.getTime()	returns a Date

Dateformat
No constructor, but a factorymethod
df.getInstance()
df.getInstance(Locale l)
df.getDateInstance()

df.getDateInstance(Style) // Style is for instance DateFormat.SHORT
df.getDateInstance(Style s, Locale l)
df.getDateInstance(Locale l)
df.getTimeInstance()
df.getTimeInstance(Locale l)

Date DateFormat.parse()	parse a string into a Date (throws a ParseException)
String format(date)	formats a date into a String

Locale

Constructor	Locale (String Language, String Country)
Constructor	Locale (String Language)

String getDisplayCountry()	returns a String representing the coutries name
String getDisplayLanguage()	returns a String representing the language name

NumberFormat
No constructor, but a factorymethod
nf.getInstance()
nf.getInstance(Locale l)
nf.getCurrencyInstance()
nf.getCurrencyInstance(Locale l)

int getMaximumFractionDigits()	retuns the maximum number of digits in the fraction
int setMaximumFractionDigits()	sets the maximum number of digits in the fraction
setParseIntegerOnly(true)	Sets whether or not numbers should be parsed as integers only
parse()	Parses text from the beginning of the given string to produce a number.

Parsing, Tokenizing and Formatting

Pattern and Matcher
example:

```
Pattern p = Pattern.compile("ab");
Matcher m = p.matcher("abaaaba");
boolean b = false;
while (m.find()) {
  System.out.print(m.start() + " ");   // prints: 0 4
}
```

A regex search runs from left to right and once a character has been used in a match it can't be reused. example: "aba" in "abababa" has two matches: 0 4

MetaCharacters

\d	a digit
\s	whitespace character
\w	a word character (numbers, letters or "_")

[abc]	searches for the characters 'a', 'b' and 'c'
[a-f,A-F]	searches for the first six characters of the alphabet (both cases)
0[xX][0-9,a-f,A-F]	searches for a zero, followed by a 'x' or 'X', followed by range of numbers and the first 6 letters of the alphabet (case insensitive)

Quantifiers

+	One or more	[1-n]
*	Zero or more	[0-n]

?	Zero or one	[0-1]
^	Negate	
.	Any character	

example:
source: “1 a12 234b”
pattern: \d+
output:
0 1
3 12
6 234

Greedy or reluctant

Greedy	Reluctant	
?	??	zero or once
*	*?	zero or more
+	+?	one or more

example:
source “yyxx.xyxx”
pattern “.”
output: 0 1 2 3 4 5 6 7 8

source “yyxx.xyxx”
pattern “\\.” // not the metacharacter . but it searches for a “.”
output: 4

Searching with a Scanner
example:

```
Scanner s = new Scanner(“ab ab abba”);
String token;
int count =1;
do {
  token = s.findInLine(\\w\\w);
  System.out.println(count + “token: “ + token);
  count++;

} while (token!=null);
```

output:
1 token: ab
2 token: ab
3 token: ab
4 token: ba
5 token: null

Tokenizing
String.split() – returns String array

example:

```
String [] token = “ab ab ab, abc, a”.split(“,”);
for (String a: token) {
   System.out.println(“>” + a + “<”);
}
```

output:
>ab ab ab<

> abc<
> a<

Tokenizing with a scanner
- Scanners can be constructed using files streams, or Strings as a source
- Tokenizing is performed in a loop so that you can exit the process at any time
- Tokens can be converted to their appropriate primitive types automatically
- The Scanner default delimiter is a whitespace
- The scanner has nextXxx() and hasNextXxx() methods for every primitive except char
- useDelimiter() method takes a String or a Pattern

example:
```
boolean b, b2;
int i;
String hits = " ";
String toBeScanned = "1 true 34 hi";
Scanner s2 = new Scanner(toBeScanned);

while (b= s2.hasNext()) {
   if (s2.hasNextInt()) {
     i = s2.nextInt();
     hits+="s";
   } else if (s2.hasNextBoolean()) {
      b2 = s2.nextBoolean();
      hits+="b";
   } else {
      s2.next();
      hits+="s2";
   }
} // hits is "sbss2"
```

Formatting with printf() and format()
printf("format string", argument(s))

format string:
%[arg_index$][flags][width][.precision]conversion

flags (5)
"-" left justify
"+" include a sign (+ or -) with this argument
"0" pad this argument with zeroes
"," use locale-specific grouping separators (i.e. the comma in 123,345)
"(" enclose negative numbers in parentheses

conversion (5)
b boolean
c char
d integer
f floating point
s string

Example:
int i1 = -123;
int i2 = 12345;

printf (">%2$b + %1$5d< \n", i1, false);

output: >false + -123<

Chapter 7 – Generics and Collections

Method of Object Class	Description
boolean equals (Object o)	Decides whether two objects are meaningful ly equivalent
void finalize()	Called by the garbage collector (when the object is not referenced anymore)
int hashcode()	Returns an int (hash) so that the object can be used in hashed Collections
final void notify()	Wakes up a thread that is waiting for this object's lock
final void notifyAll()	Wakes up all threads that are waiting for this object's lock
final void wait()	Causes this thread to wait until another thread calls notify or notifyAll on this object
String toString()	Returns a string representation of this object

The equals contract

1. It is reflexive. For any reference variable x. x.equals(x) should return true.
2. It is symmetric. For any reference variable x, y: x.equals(y) should return true if and only if y.equals(x) returns true
3. It is transitive. For any reference variable x, y and z: If x.equals(y) returns true and y.return(z) returns true, then x.return(z) must return true
4. It is consistent. For any reference variable x, y: Multiple invocations of x.equals(y) consistently return true or return false, provided no information used in the equal comparison on the object has changed
5. For any non-null reference variable x: x.equals(null) should return false

Hashing

Hashing is a 2-step process
1) Find the right bucket, using the hashcode() method
2) Search the bucket for the right element, using the equals() method

The hashCode contract

1) Whenever it is invoked on the same object more than once during an execution of a Java application, the hascode() method must consistently return the same integer, provided no no information used in the equals() comparisons on the object is modified.
2) If two objects are equal according to the equals (object) method, then calling the hashcode() method on each of the two objects must produce the same integer result.
3) It is not required that if two objects are considered unequal according to the equals() method, then calling the hascode() method on each of the two objects must produce the distinct integer results

Condition	Required	Not Required (but allowed)
x.equals(y) == true	x.hashcode() == y.hashcode()	
x.hashcode() == y.hashcode()		x.equals(y) == true
x.equals(y) == false		no hashcode requirement
x.hashcode() != y.hashcode()	x.equals(y) == false	

Dont use transient variables in hashcode() methods

Collections

Key interfaces of the Collections Framework

1) Collection
2) Set
3) SortedSet
4) NavigableSet
5) List
6) Map
7) SortedMap
8) NavigableMap
9) Queue

Key implementation classes

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

Basic Collection Flavours

1) Lists – List of things (classes that implement List)
2) Sets – Unique things (classes that implement Set)
3) Maps – Things with an unique ID (classes that implement Map)
4) Queues – Things arranged by the order in which they are to be processed

Ordered – You can iterate through a specific (not random) order

Sorted – The order in the collection is determined according to some rule or rules known as the sort order. A sorted collection uses the compareTo() method during insertion

Sorted Collections

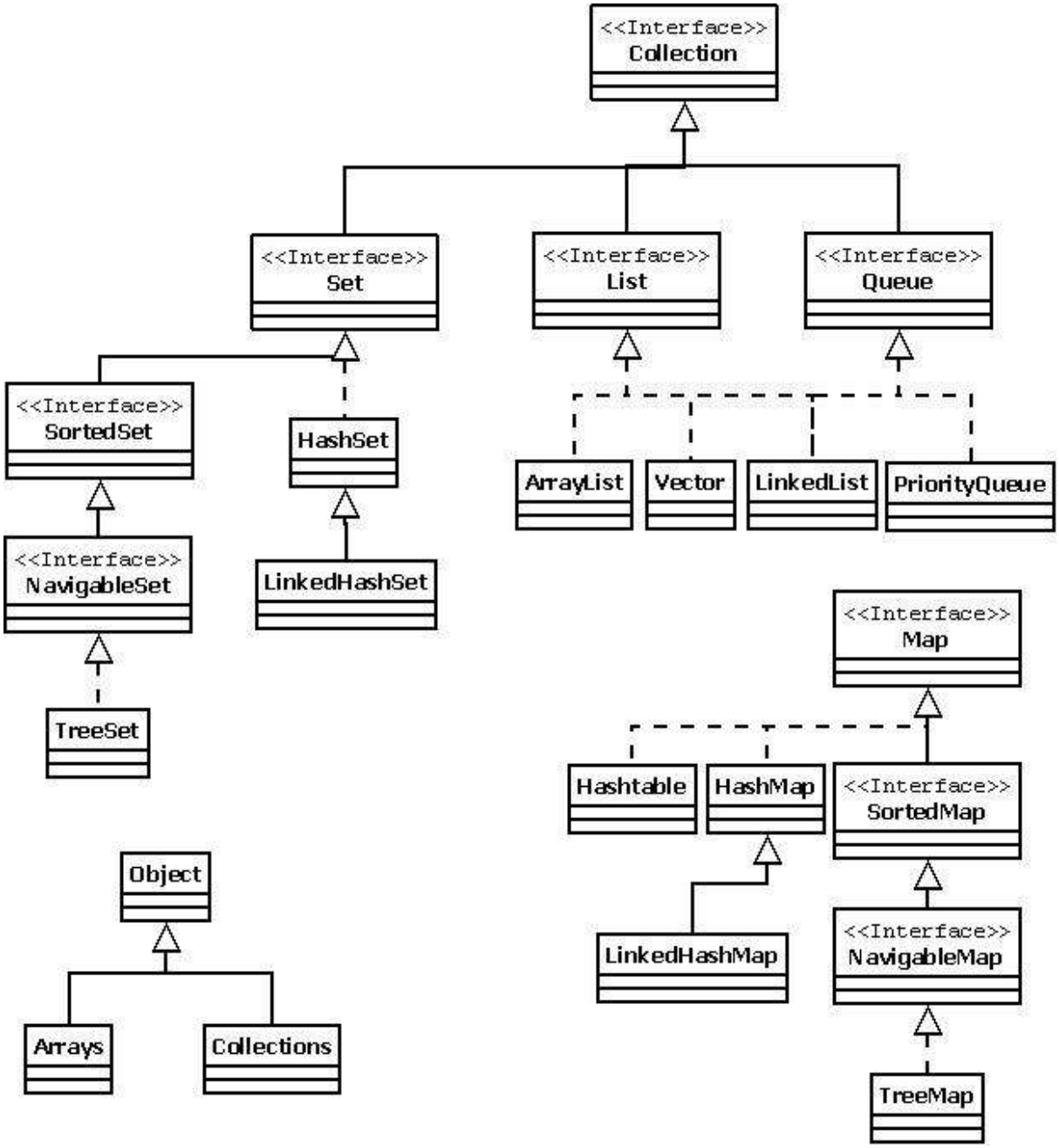
TreeMap	By natural order or custom comparison rules (*uses compareTo() method*)
TreeSet	By natural order or custom comparison rules (*uses compareTo() method*)
PriorityQueue	By to-do order

Ordered Collections

LinkedHashMap	By insertion order or last access order
LinkedHashSet	By insertion order
ArrayList	By index
Vector	By index
LinkedList	By index

Unordered Collections

HashMap
Hashtable
HashSet



11 Classes & 9 Interfaces & 2 Utility Classes

The Comparable Interface

The interface is used by

1) Collections.sort
2) Arrays.sort

implement the following method:
public int thisObject.compareTo(anotherObject)

The int returned by the compareTo() method is:

negative	if thisObject < anotherObject
zero	if thisObject == anotherObject
positive	if thisObject > anotherObject

Overriding compareTo and equals

When you override equals you must take an object as an argument
When you override compareTo you should take the object type you are sorting (object is allowed):
example:

```
class DVDinfo implements Comparable<DVDInfo>{
   public int compareTo (DVDInfo d){}
}
```

The Comparator interface

The interface is used by
1) Collections.sort
2) Arrays.sort

implement the following method:
public int compare (thisObject, anotherObject)

The int returned by the compareTo() method is:
negative if thisObject < anotherObject
zero if thisObject == anotherObject
positive if thisObject > anotherObject

java.lang.Comparable	java.util.Comparator
int thisObject.compareTo(anotherObject)	int compare(thisObject, anotherObject)
You must modify the class whose instances you want to sort	You build a seperate class from the class whose instances you want to sort
One sort sequence	Many sort sequences (by creating many comparators)
Implemented frequently in the API by: String, Wrapper Classes, Date, Calendar...	Meant to be implemented to sort instances of third-party classes

Searching Arrays and Collections

- Searches are performed using the binarySearch() method
- Successful searches return the int index of the element being searched
- Unsuccessful searches return an int index that represents the insertion point. The insertion point is the place in the collection/array where the element would be inserted to keep the collection/array properly sorted. The insertion point formula is **(– (insertion point) – 1)**
- The collection/array being searched has to be sorted, otherwise the outcome will be unpredictable
- If the collection/array was sorted in natural order you can't use a Comparator to search
- If the collection/array was sorted with a comparator, it has to be searched with a Comparator

Converting Arrays to Lists and Lists to Arrays

example Array to List

```
String[] nummers = {"one", "two", "three"};
List asList = Arrays.asList(nummers);
```

example List to Array

```
List<Integer> lijst = new ArrayList<Integer>();

Object[] Oarray = lijst.toArray();  // Object Array

Integer[] ia = new Integer[2];
ia = lijst.toArray(ia);       // Integer Array
```

Generic Iterator (no cast required)

```
List<Integer> lijst = new ArrayList<Integer>();
Iterator<Integer> it = lijst.iterator();

if (it.hasNext()) Integer i1 = it.next();
```

Method Overview for Arrays and Collections

java.util.Arrays	Description
static List asList(T[])	Convert an array to a list (and bind them)
static int binarySearch(Object[], key) static int binarySearch(primitive[], key)	Search a sorted array for a given value, return an index or an insertion point
static int binarySearch(T[], key, Comparator)	Search a Comparator-sorted array
static boolean equals(Object[], Object[]) static boolean equals(primitive[], primitive[])	Compare two arrays and determine if their contents are equal
public static void sort(Object[]) public static void sort(primitive[])	Sort the elements of an array by natural order
public static void sort(T[], Comparator)	Sort the elements of an array using a Comparator
public static String toString(Object[]) public static String toString(primitive[])	Create a string containing the elements of an array

java.util.Collections	Description
static int binarySearch(List, key) static int binarySearch(List, key, Comparator)	Search a sorted list for a given value return an index or an insertion point
static void reverse(List)	Reverse the order of the elements of the list
static Comparator reverseOrder() static Comparator reverseOrder(Comparator)	Return a Comparator that sorts the reverse of the collection's current sort sequence
static void sort(List) static void sort(List, Comparator)	Sort a List by natural order or by Comparator

Method Overview for List, Set, Map and Queue

Key Interface methods	List	Set	Map	Description
boolean add(element) boolean add(element, index)	X X	X		Add an element. For a List optionally at a given index
boolean contains(object) boolean containsKey(object key) boolean containsValue(object value)	X	X	 X X	Search a collection for an object (or optionally for a Map a key) return the result as a boolean
Object get(index) Object get(key)	X		 X	Get an object from a collection via an index or a key
int indexOf(Object)	X			Get the location of an Object in a List
Iterator iterator()	X	X		Get an iterator for a List or a Set
Set keySet()			X	Return a Set of keys of the Map
put(key, value)			X	Add a key, value pair to a Map
remove(index) remove(object) remove(key)	X X	 X	 X	Remove an element via an index, or via the elements value or via a key
int size()	X	X	X	Return the number of elements of a collection
Object[] toArray() T[] toArray(T[])	X	X		Return an array containing the elements of the collection
Collection values()			X	Returns a collection with the values from the map

Method Overview for PriorityQueue

Method	Description
offer()	Add an object to the queue
peek()	Retrieves the element at the head of the queue
poll()	Retrieves and removes the element at the head of the queue

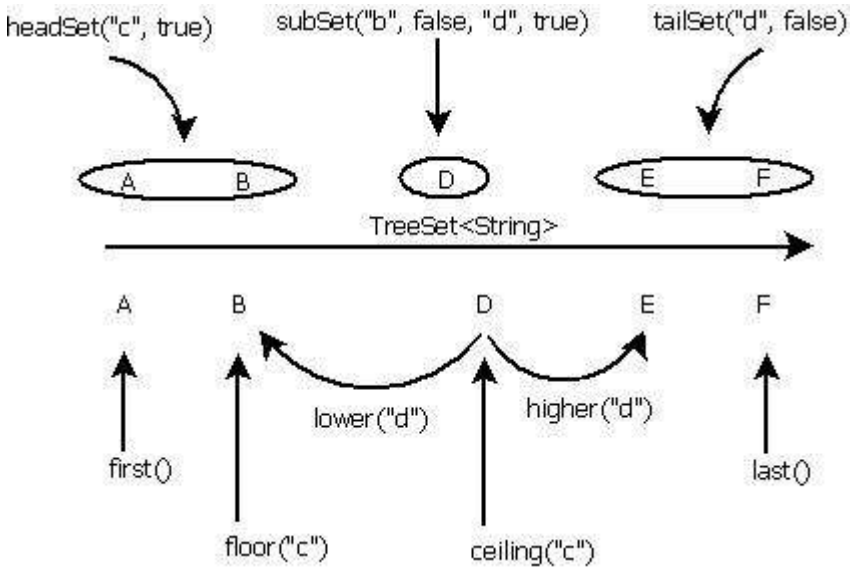
Method Overview for SortedSet

Method	Description
Comparator<? super E> comparator()	Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
E first()	Returns the first (lowest) element currently in this set.
E last()	Returns the last (highest) element currently in this set.
SortedSet<E> headSet(E toElement)	Returns a view of the portion of this set whose elements are strictly less than toElement.
SortedSet<E> tailSet(E fromElement)	Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.

Method Overview for NavigableSet

example:

```
public class SortedSetMap {
    private NavigableSet<String> alphaLijst = new TreeSet<String>();
    public SortedSetMap(){
      fillLijst();
    }
    public NavigableSet<String> getAlphaLijst() {
            return alphaLijst;
    }
    public void setAlphaLijst(NavigableSet<String> alphaLijst) {
            this.alphaLijst = alphaLijst;
    }
    private void fillLijst (){
            alphaLijst.add("E");
            alphaLijst.add("A");
            alphaLijst.add("B");
            alphaLijst.add("D");
            alphaLijst.add("F");
    }
}
```



Method	Description
Iterator<E> descendingIterator()	Returns an iterator over the elements in descending order
NavigableSet<E> descendingSet()	Returns a reverse order view of the elements in this set
E ceiling(E e)	Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
E higher(E e)	Returns the least element in this set strictly greater than the given element, or null if there is no such element.
E lower(E e)	Returns the greatest element in this set strictly less than the given element, or null if there is no such element.
E floor(E e)	Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
E pollFirst()	Retrieves and removes the first (lowest) element, or returns null if this set is empty.
E pollLast()	Retrieves and removes the last (highest) element, or returns null if this set is empty.
NavigableSet<E> headSet(E toElement, boolean inclusive)	Returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement
NavigableSet<E> tailSet(E fromElement, boolean inclusive)	Returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement
NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	Returns a view of the portion of this set whose elements range from fromElement to toElement.

Method Overview for NavigableMap

Method	Description
NavigableMap<K,V> descendingMap()	Returns a reverse order view of the mappings contained in this map
NavigableSet<K> descendingKeySet()	Returns a reverse order NavigableSet view of the keys contained in this map.
NavigableSet<K> navigableKeySet()	Returns a NavigableSet view of the keys contained in this map.
NavigableMap<K,V> headMap(K toKey, boolean inclusive)	Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey
NavigableMap<K,V> tailMap(K	Returns a view of the portion of this map whose keys are

fromKey, boolean inclusive)	greater than (or equal to, if inclusive is true) fromKey
SortedMap<K,V> subMap(K fromKey, K toKey)	Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.
firstEntry Map.Entry<K,V> firstEntry()	Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V> pollFirstEntry()	Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
Map.Entry<K,V> lastEntry()	Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
Map.Entry<K,V> pollLastEntry()	Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
K floorKey(K key)	Returns the greatest key less than or equal to the given key, or null if there is no such key.
K ceilingKey(K key)	Returns the least key greater than or equal to the given key, or null if there is no such key.
K higherKey(K key)	Returns the least key strictly greater than the given key, or null if there is no such key.
K lowerKey(K key)	Returns the greatest key strictly less than the given key, or null if there is no such key.
Map.Entry<K,V> floorEntry(K key)	Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
Map.Entry<K,V> ceilingEntry(K key)	Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
Map.Entry<K,V> higherEntry(K key)	Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.
Map.Entry<K,V> lowerEntry(K key)	Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.

Generic Types

Generic Collection

```
List<String> myList = new ArrayList<String>()
```

Generic method parameter

```
void takeListOfStrings(List<String> strings){
   strings.add("String");
   strings.add(new Integer(34)); // compiler error
}
```

List, ArrayList -> basic type of the collection
<String> -> generic type of the collection

- If you add anything to a typed collection other than the generic type you will get a compile error
- If you remove something from the collection, you don't need a cast
- With arrays there is a runtime Exception – ArrayStoreException if you put the wrong thing in an array

A generic Iterator

```
List<Transaction> myList;
Iteractor<Transaction> i = myList.iterator();
```

A generic Comparator

```
public class CompareTransaction implements Comparator<Transaction> {
      public int compare (Transaction t1, Transaction t2){
      }
}
```

Mixing Generic code with non generic code

- It is possible to pass a typed collection to an old non-generic method
- It is possible to pass a typed collection and add something to it via an old non-generic method
- Using a non-generic method *compiles with warnings*

Polymorphism and Generics

The base type can be use polymorphically:

```
List<String> myList = new ArrayList<String>()
```

because List is a supertype of ArrayList

The generic type *cannot* be use polymorphically

```
List<Animal> myAnimal = new ArrayList<Dog>()   // NOT ALLOWED
```

You are able to put subtypes of the generic type into a generic collection:

```
List<Animal> myAnimal = new ArrayList<Animal>()
myAnimal.add(new Dog());
myAnimal.add(new Cat());
```

List <? extends Animal> lijst

lijst can be assigned a collection that is a subtype of List and typed for Animal or anything that extends Animal, but nothing can be added to the collection.

```
public class AnimalDoctorGeneric {
      public readAnimal(List<? extends Animal> lijst){
         Animal a = lijst.get(0);
         System.out.println("Animal: " + a);
      }

      public static void main(String args[]) {
         AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
         List<Dog> myList = new ArrayList<Dog>();
         myList.add(new Dog());
         doc.readAnimal(myList);
      }
}
```

List <? super Dog> lijst

lijst can be assigned any List with a generic type that is of type Dog or a supertype of Dog. You can add objects to the list but only of type Dog.

```
public class AnimalDoctorGeneric {
      public void addAnimal(List<? super Dog> lijst) {
            lijst.add(new Dog());
      }

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

```
            AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
            List<Animal> myList = new ArrayList<Animal>();
            myList.add(new Dog());
            doc.addAnimal(myList);
            List<Object> myObjectList = new ArrayList<Object>();
            myObjectList.add(new Dog());
            myObjectList.add(new Object());
            doc.addAnimal(myObjectList);
        }
}
```

- List<?> and List<? extends Object> are identical
- Wildcards can only be used in reference declarations:
  - List<?> lijst = new ArrayList<Dog>();
  - List<? extends Animal> lijst = new ArrayList<Dog>();
  - List<? super Dog> lijst = new ArrayList<Animal>();
  - List<?> lijst = new ArrayList<**? extends Animal**>(); // NOT ALLOWED in object creation

Generic Declarations

public interface List**<E>**
    -> E means “Element” used for Collections
public class RentalGeneric**<T>**
    -> T means “Type” and is used for anything other than Collections
public class AnimalHolder**<T extends Animal>**
    -> specify a range of the parameter T (it cannot be an Integer)
public class UseTwo**<X, Y>**
    -> use more than one generic type
public **<T>** void makeArrayList(**T** t)
    -> A generic method: declare the generic type before the return type
public **<T extends Animal>** void makeArrayList(**T** t)
    -> A generic method with boundaries on the type
public **<T>** radio(**T** t)
    -> a constructor defined in a generic way
<List<List<Integer>> table = new ArrayList**<List<Integer>>**
    -> a list that contains a lists of Integers

Chapter 8 – Inner Classes

There are four different Inner classes:

1) Regular Inner Class
2) Static Inner Class
3) Method Local Inner Class
4) Anonymous Inner Class

Regular Inner Class

example:

```
class MyOuter {
   private int x = 7;

   class MyInner {
      public void seeOuter() {
      System.out.println("Outer x is: " + x);
      } // end method seeOuter
    } // end class MyInner
} // end class MyOuter
```

Instantiating from within the outer class (via a method on the outer instance):

```
class MyOuter {
   private int x = 7;

   class MyInner {
      public void seeOuter() {
      System.out.println("Outer x is: " + x);
      } // end method seeOuter
    } // end class MyInner

   public void makeInner(){
      MyInner in = new MyInner();
      in.seeOuter
   }
} // end class MyOuter
```

Instantiating from outside the outer class instance code

MyOuter.java

```
public class MyOuter {
      private int x = 7;

      class MyInner {
         public void seeOuter() {
         System.out.println("Outer x is: " + x);
         } // end method seeOuter
       } // end class MyInner
   } // end class MyOuter
```

Inner.Outer.java

```
public class InnerOuter {

public static void main (String[] args) {
   MyOuter out = new MyOuter();
```

```
   MyOuter.MyInner in = out.new MyInner();
   MyOuter.MyInner inOneLine = new MyOuter().new MyInner();
   in.seeOuter();
   inOneLine.seeOuter():
}
```

Referencing the Inner or Outer instance from within the inner class

```
class MyOuter {
   private int x = 7;

   class MyInner {
      public void seeOuter() {
      System.out.println("Outer x is: " + x);
      System.out.println("Inner reference is: " + this);
      System.out.println("Outer reference is: " + MyOuter.this);

      } // end method seeOuter
    } // end class MyInner
} // end class MyOuter
```

Allowed modifiers on a inner class

1) final
2) abstract
3) public
4) private
5) protected
6) static (= static nested class)
7) strictfp

Method Local Inner Class

example:

```
class MyOuter {
   private int x = 7;
   void doStuff() {
      class MyInner {
         public void seeOuter() {
         System.out.println("Outer x is: " + x);
       } // end class MyInner

       MyInner inner = new MyInner();
       inner.seeOuter();

    } // end of doStuff()

} // end class MyOuter
```

- Method Local Inner Class cannot use (non-final) local variables of the method (stack versus heap)
- Method Local Inner Class can use **final** local variables
- A Method Local Inner Class defined in a **static** method has only access to static members

Anonymous Inner Class (can even be defined in an argument of a method)

There are two different flavors:

example flavor one:

```
class Popcorn {
   public void pop() {
      System.out.println("popcorn");
   }
}
class Food {
   Popcorn p = new Popcorn () {
      public void pop() {
         System.out.println("anonymous popcorn");
      }
   }; // close with SEMICOLLON
}
```

- The Popcorn reference variable refers not to an instance of Popcorn, but to an instance of an anonymous (unnamed) subclass of Popcorn.

```
example flavor two:
interface Cookable {
   public void cook()
}
class Food {
   Cookable c = new Cookable () {
      public void cook() {
         System.out.println("anonymous cookable implementer");
      }
   }; // close with SEMICOLLON
}
```

- The Cookable reference variable refers not to an instance of Cookable, but to an instance of an anonymous (unnamed) implementation of the interface Cookable

Static Nested Classes

example:

```
class BigOuter {
   static class Nest { void go() { System.out.println("hi"); } }
}
class Broom {
   static class B2 { void goB2() { System.out.println("hi2"); } }
   public static void main (String[] args) {
      BigOuter.Nest n = new BigOuter.Nest();
      n.go();
      B2 b2 = new B2();
      b2.go();
   }
}
```

Chapter 9 – Threads

Defining and starting:

1. Extend the Thread class
2. Override the public void run() method

Methods Thread Class

1) sleep() (Static)

- slows down a thread to let it sleep for X miliseconds
- after the sleep period expires it doesn't mean that it will start running immediatly (Runnable state)

2) yield() (Static)

- make the current running thread go back to Runnable and let other threads with equal priority do their job

3) join()

- Blocks the current running thread until this one (the one joining) has finished.
- If called from the main() method it will block main() until the one joining is finished.

4) setPriority()

- sets the priority of the thread (Thread.MIN_PRIORITY,Thread.NORM_PRIORITY,Thread.MAX_PRIORITY)
- if not set explicitly, then the thread will get the same priority as the one starting it

5) start()

- starts a thread

6) interrupt()

- Calling interrupt on a thread will cause an interruptedException only if the thread on which it is called is blocked because of :
- wait()
- join()
- sleep()

Deamon Thread

- A thread is either a user thread or a deamon thread. t.setDeamon(true); creates a deamon thread
- steDeamon has to be called before the thread is started
- The JVM exits if all running threads are deamon threads

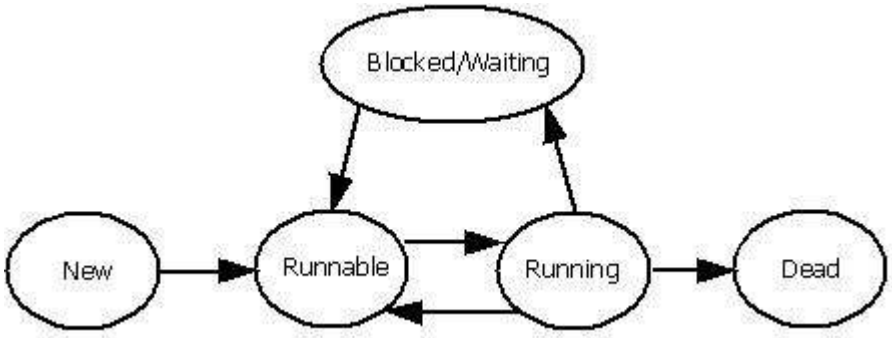
Methods of the Object class

Can only be used from a synchronized context (otherwise IllegalMonitorStateException)

1) wait()
2) notify()
3) notifAll()

If the monitor object is not explicitly named, it will be *this()* object

States of a Thread



Synchronizing Code

- Regulate concurrent access
- Only methods and blocks can be synchronized: not variables
- Each object has one lock
- Not all the methods need to be synchronized
- If a thread goes to sleep() it keeps all the locks
- Calling join() and yield() will keep the locks
- Calling notify() or notifyAll() keeps the lock until the synchronized code has finished!
- Calling wait() gives up the lock on the monitor object
- A thread can acquire more than one lock
- A static method can be synchronized using the class lock: synchronized(MyClass.class)
- A synchronized run() method (Thread object or a class that implements the Runnable interface) is only useful if the same instance is used in 2 or more threads

Locking

- Threads calling non-static synchronized methods in the same class will only block each other if they are invoked using the same instance. They lock on the 'this' instance, so if called on different instances they will get two different locks which do not interfere with each other
- Threads calling static synchronized methods in the same class will always lock each other
- A static synchronized method and a non-static synchronized method will never block each other (one on a object-instance and one on the class-instance)

Thread Safe

- Watch out with class that has thread-safe methods: each individual method is thread-safe but calling two methods in a row aren't

Chapter 10 – Development

Java's compiler

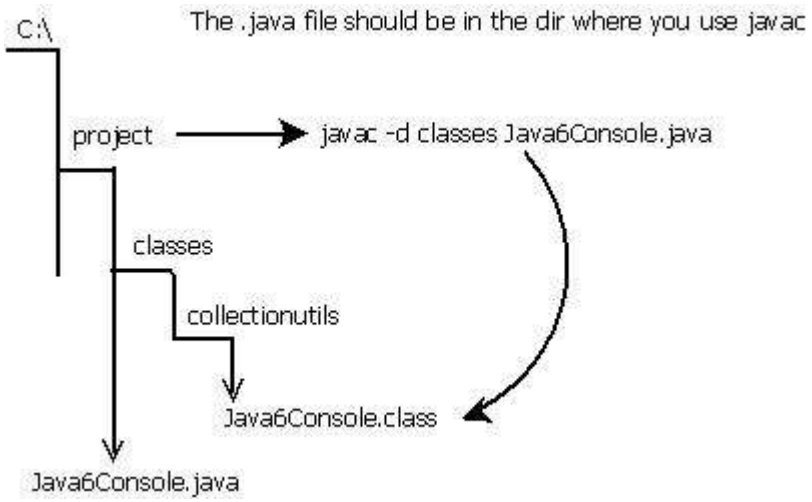
javac [options] [source files]

javac -d
destination of the .class files

- from the package name of the .java file it can create the correct directory structure
- if the destination directory doesn't exist it will produce a compiler error
- specify .java

example:

```
package collectionutils;
public class Java6Console {}
```



Java command line

java [options] class [args]

- specify one class file but don't put the .class

java -D

java -DcmdProp=cmdVal is adding an extra system property. (use the getProperty to get it)

java -DcmdProp=cmdVal TestProps x q is adding a system property and passing arguments x and 1 to TestProps

System.getProperty("FLAG"); // returns the value of the system property

System.getProperty("FLAG", "false"); // returns the value of the system prop. and if it doesn't exist false

Valid main() declarations:

- static public void main(String[] args)
- public static main void(String... x)
- public static main void(String bla_bla[])

Java search algorithm

Both java and javac use the same basic search algorithm:

- They both have the same list of places (directories) they search, to look for classes.
- They both search through this list of directories in the same order.
- As soon as they find the class they're looking for, they stop searching for that class.
- In the case that their search lists contain two or more files with the same name, the first file found will be the file that is used.
- The first place they look is in the directories that contain the classes that come standard with J2SE.
- The second place they look is in the directories defined by classpaths.
- Classpaths should be thought of as "class search paths" They are lists of directories in which classes might be found.
- There are two places where classpaths can be declared:
- A classpath can be declared as an operating system environment variable. The classpath declared here is used by default, whenever java or javac are invoked.
- A classpath can be declared as a command-line option for either java or javac. Classpaths declared as command-line options override the classpath declared as an environment variable, but they persist only for the length of the invocation.

java -classpath (or -cp)

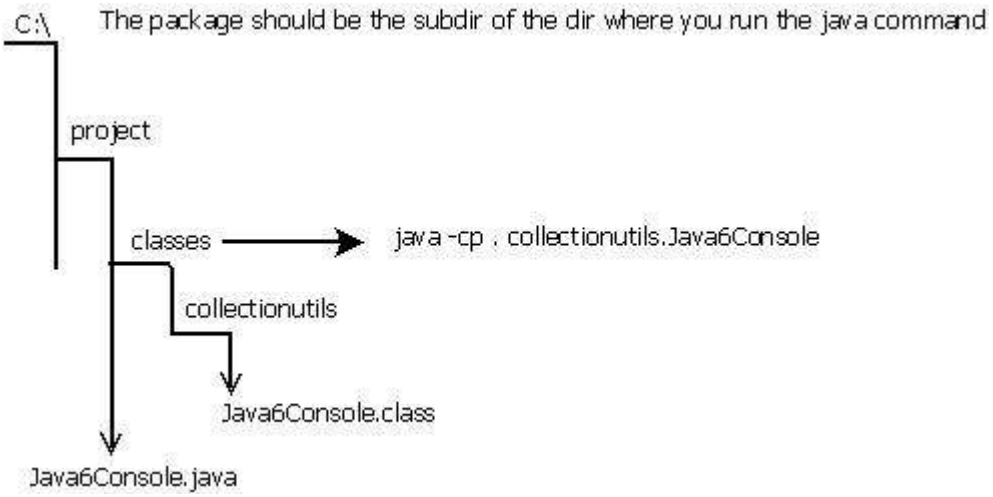
When a class file is defined in a package the fully qualified classname (fqn) consists of the package name.

example:

```
package collectionutils;
public class Java6Console {}
```

fqn = `collectionutils.Java6Console`

In order to run the `Java6Console` it has to have the **package** root dir as a subdir.

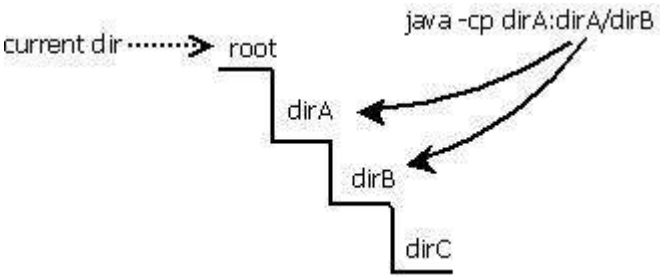


Absolute and Relative paths

Absolute path starts with an / (unix) or c:\ (windows)

If the directory tree is (root)/dirA/dirB/dirC and java -cp dirA:dirA/dirB and the current dir is:

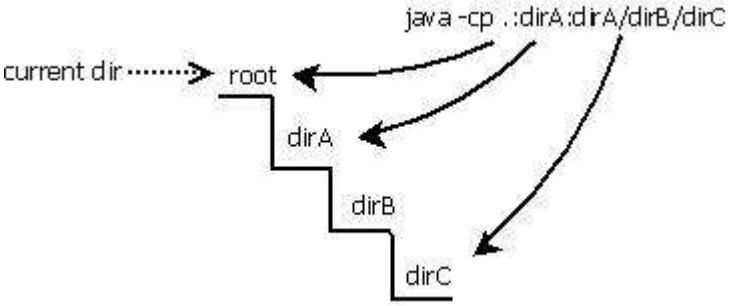
- (root), then dirA & dirB are searched for class files



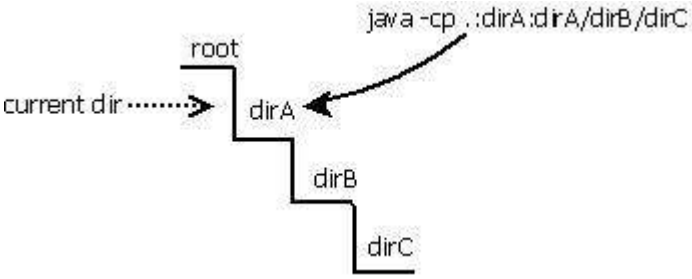
- dirA, then no directories are searched

If the command is java -cp .:dirA:dirA/dirB/dirC and the current dir is:

- (root), then (root), dirA and dirC are searched

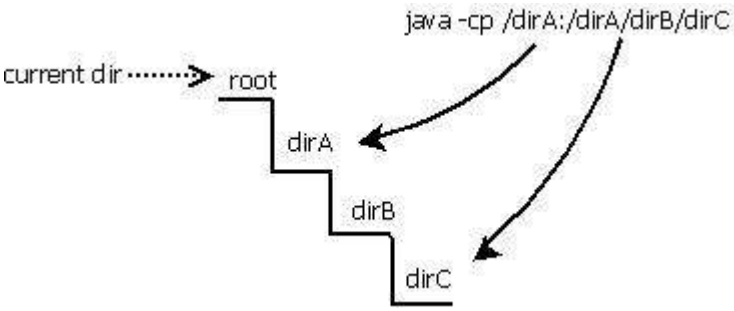


- dirA, then only dirA is searched (because of the ".", meaning current dir)

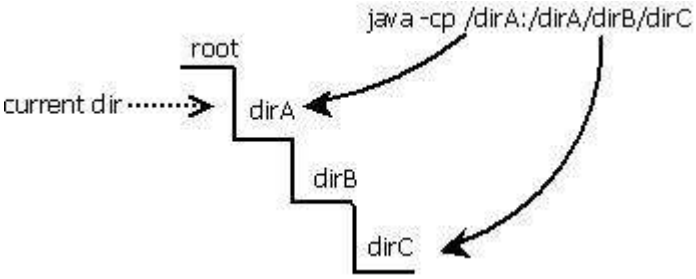


If the command is java -cp /dirA:/dirA/dirB/dirC and the current dir is:

- (root), the path is absolute so dirA and dirC are searched



- dirA, the path is absolute so dirA and dirC are searched



Jar files

create a jar file: jar -cf MyJar.jar myApp (it will take the myApp dir and all subdirs)

read a jar file jar -tf MyJar.jar

example (TestProps uses the class TestJar)

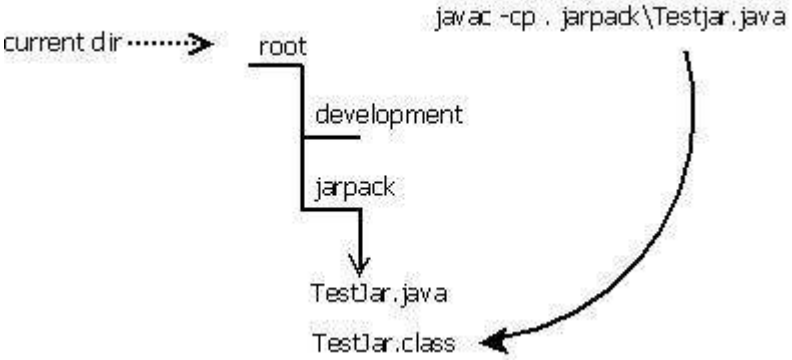
(root)\development\TestProps.java

```
package development;
import jarpack.TestJar;
public class TestProps {
    public static void main(String[] args) {
          TestJar tj = new TestJar();
          System.out.println(tj.getDateAsString());
    }
}
```

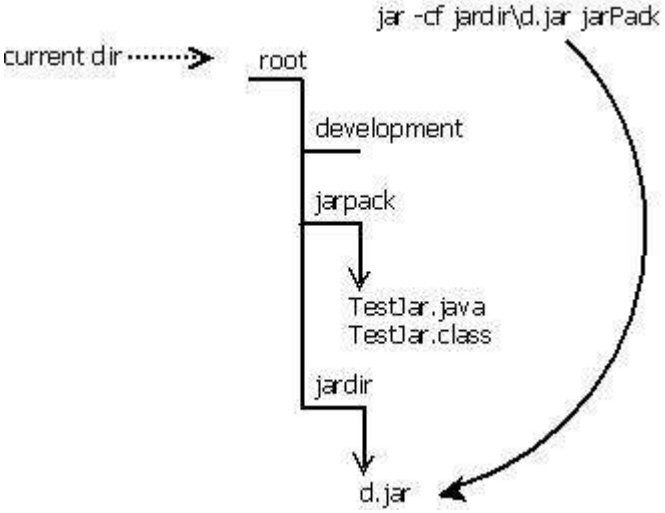
(root)\jarpack\TestJar.java

```
package jarpack;
import java.text.DateFormat;
import java.util.Date;
public class TestJar {
    public String getDateAsString(){
          DateFormat df = DateFormat.getDateInstance();
          return df.format(new Date());
    }
}
```

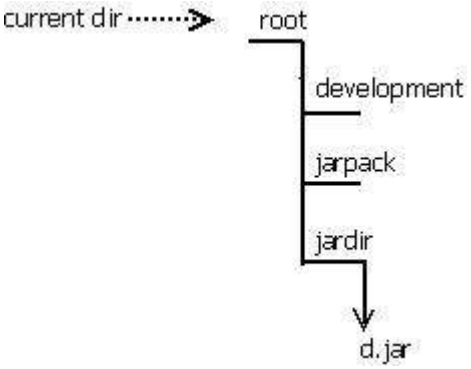
compile TestJar: javac -cp . jarpack\TestJar.java



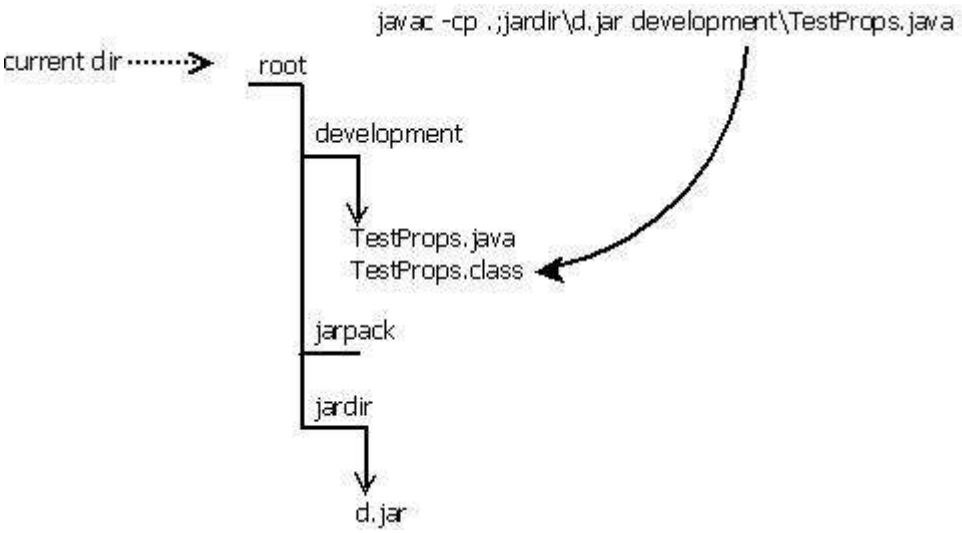
create jar TestJar: jar -cf jardir\d.jar jarpack



Remove the files in the jarpack directory (just for the sake of the example)



compile TestProps: javac -cp .;jardir\d.jar development\TestProps.java



(leaving out classpath entry to d.jar gives a compile error)

run TestProps: java -cp .;jardir\d.jar development.TestProps

(leaving out classpath entry d.jar returns java.lang.NoClassDefFoundError)

Static Imports

import static java.lang.Integer.*;

Then in the code you can use system.out.println(MAX_VALUE) instead of (Integer.MAX_VALUE)

- use import static
- import of static object references, constants, and static methods