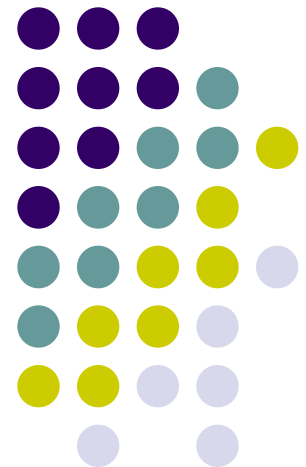
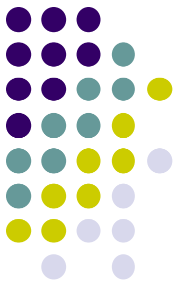
Advanced programming in Java

Ing. Marek Běhálek
katedra informatiky FEI VŠB-TUO
A-1018 / 597 324 251
http://www.cs.vsb.cz/behalek
marek.behalek@vsb.cz

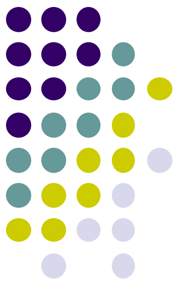


Overview



- Generics
- Java Collections Framework
- Reflection
- Annotations
- Serializations
- Streams in Java
- Thread and Synchronization

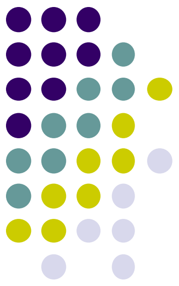
Generics - Compare



```
List li = new ArrayList();
li.add(new Integer(1));
Integer x = (Integer)li.get(0);

List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(1));
Integer x = li.get(0);
```

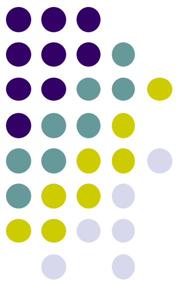
- The main point: “old” containers hold “Object” objects and need casts which are problematic because:
- Cast is something the programmer thinks is true at a single point.
- Generic type is true everywhere.

Generics - What Generics in Java are?



- A way to control a class type definitions.
- Otherwise known as *parameterised types* or *templates*.
- A way of improving the clarity of code
- A way of avoiding `(casts)` in code, turning run-time errors (typically **ClassCastException**) into compile-time errors. This is A Good Thing.
- Benefits of generic types
  - increased expressive power
  - improved type safety
  - explicit type parameters and implicit type casts
- Only in Java 5 and above

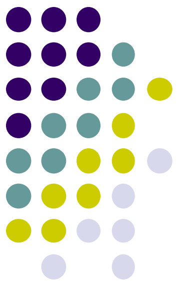
Generics - Definition of Generics



```
interface Collection<A> {
        public void add (A x);
        public Iterator<A> iterator ();
}
class LinkedList<A> implements Collection<A> {
        protected class Node {
                A elt;
                Node next = null;
                Node (A elt) { this.elt = elt; }
        }
        ...
}
```

- type variable = *"placeholder"* for an unknown type
  - similar to a type, but not really a type
  - several restrictions
    - not allowed in new expressions, cannot be derived from, no class literal, ...

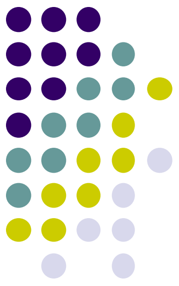
Generics - Type parameter bounds



```
public interface Comparable<T> { public int compareTo(T arg); }

public class TreeMap<K extends Comparable<K>,V> {
        private static class Entry<K,V> { ... }
        private Entry<K,V> getEntry(K key) {
                while (p != null) {
                        int cmp = k.compareTo(p.key);
                ... }
        ...
        }
...
```

- bounds = super-type of a type variable
  - purpose: make available non-static methods of a type variable
  - limitations: gives no access to constructors or static methods

Generics - Generics and sub-typing

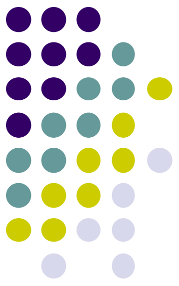


- Should this be valid?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
//…
lo.add(new Object());
String s = ls.get(0);
```

- **In other words: is List<String> a subtype of List<Object> ?**
- **The answer is NO!**
- But inheritance is a powerful tool, and we want to use it with generics…

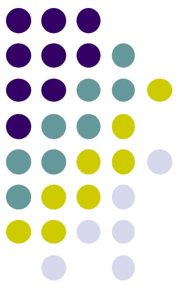
Generics - Example: Statistics class



- This class gets an array of numbers and calculates their average.

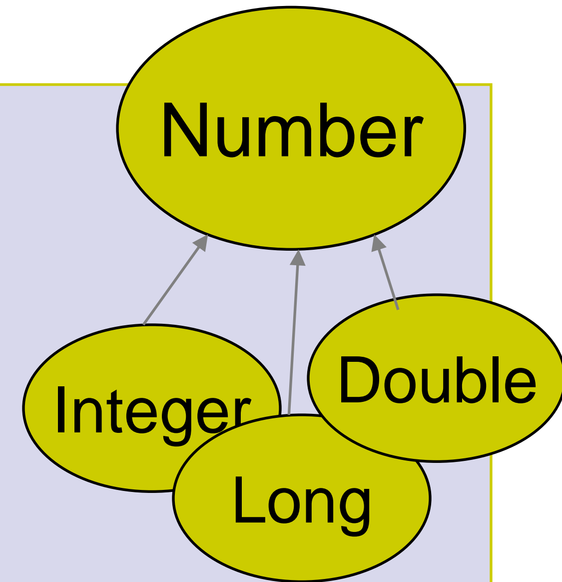
```
public class Stats<T> {
      T[] nums; // nums is an array of type T

      Stats(T[] o) {
        nums = o;
      }

      // Return type double in all cases.
      double average() {
        double sum = 0.0;

        for(int i=0; i < nums.length; i++)
          sum += nums[i].doubleValue(); // Error!!!

        return sum / nums.length;
      }
    }
```

Generics - Example: Statistics class

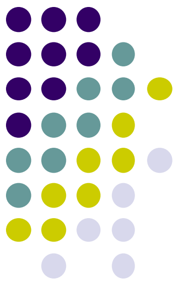


- To solve the problem we will use a bounded type.

```
public class Stats <T extends Number> {
    T[] nums; // nums is an array of type T

    Stats(T[] o) {
      nums = o;
    }

    // Return type double in all cases.
    double average() {
      double sum = 0.0;

      for(int i=0; i < nums.length; i++)
        sum += nums[i].doubleValue(); // now it's OK.

      return sum / nums.length;
    }
  }
```

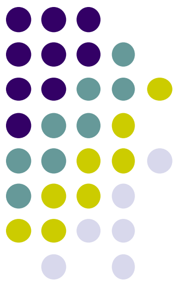


Generics - Using generic types (1)



- Can use generic types with or without type argument specification
  - with concrete type arguments
    - concrete instantiation
  - without type arguments
    - raw type
  - with wildcard arguments
    - wildcard instantiation

Generics - Using generic types (2)

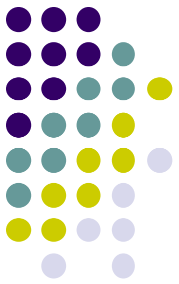


- Concrete instantiation
  - type argument is a concrete type

```
void printDirectoryNames(Collection<File> files) {
   for (File f : files)
                 if (f.isDirectory())
                             System.out.println(f);
}
```

  - more expressive type information
    - enables compile-time type checks
- Raw type
  - no type argument specified
  - permitted for compatibility reasons
    - permits mix of non-generic (legacy) code with generic code

Generics - Wildcards (1)



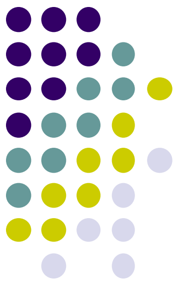
- What is the problem with this code?

```
void printCollection( Collection<Object> c){
        for (Object o : c)
                System.out.println(o);
}
```

- Collection<Object> is NOT a supertype of any other collection.
    - this code is not so usefull…
- The solution: wildcards:

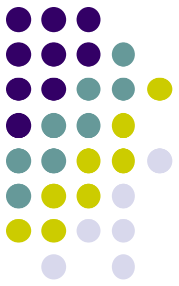
```
void printCollection( Collection<?> c){
        for (Object o : c)
                System.out.println(o);
}
```

Generics - Wildcards (2)



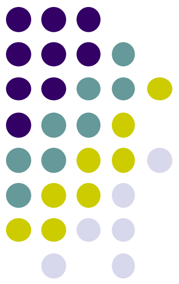
- A wildcard denotes a representative from a family of types
  - unbounded wildcard - ?
    - all types
  - lower-bound wildcard - ? extends Supertype
    - all types that are subtypes of Supertype
  - upper-bound wildcard - ? super Subtype
    - all types that are supertypes of Subtype

Generics - Bounded wildcard, Stats revisited



```
public class Stats{

  static double average(List<? extends Number> nums) {
    double sum = 0.0;

    for (Number num : nums)
      sum += num.doubleValue();

    return sum / nums.size();
  }

  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    //List<Number> li1 = Arrays.asList(inums); //compilation error
        //List<? extends Number> li2 = Arrays.asList(inums);  //ok
    List<Integer> li = Arrays.asList(inums);
    System.out.println(average(li)); //prints 3.0  }
}
```

Generics - Generic methods



- the avarage() method signature:

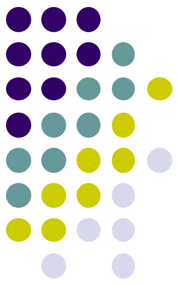
```
static double average(List<? extends Number> nums)
```

- An alternative (equivalent) signature:

```
static <T extends Number> double average(List<T> nums)
```

- The later is called a generic method.
- Which is better?
  - When there are no dependencies between the method parameters - use wildcards.

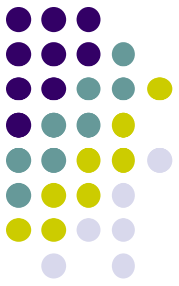
Generics - Calculating the median



```
public class Stats{

  static double average(List<? Extends number> nums) { …}

  static <T extends Number> T median(List<T> nums) {
    int pos = nums.size()/2;
    return nums.get(pos);
  }

  public static void main(String args[]) {
       Integer inums[] = { 0, 0, 0, 0, 100};
       List<Integer> li = Arrays.asList(inums);
       System.out.println(average(li));
       System.out.println(median(li));
  }
}
```

This way the compiler knows about the dependency between the input and output arguments.

Generics - Another generic method examples



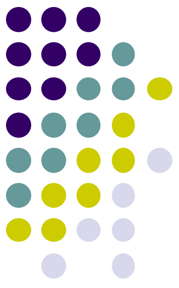
Determine if an object is in an array:

```
static <T, V extends T> boolean isIn(V x, T[] y) {

  for(int i=0; i < y.length; i++)
    if(y[i].equals(x)) return true;

  return false;
}
```

Collections.sort()

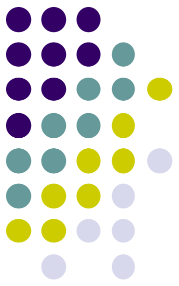
```
public static <T extends Comparable<? super T>> void sort(List<T> list) {
…
}
```

Generics - Java Generics Implemetation



- There are two general approaches:
  - *Code specialisation* – generate a version of the class for each way it's used (what C++ does)
  - *Code sharing* – use a single version of the class for all uses, but perform checks as each use occurs (what Java does)
- The Java compiler uses ***type erasure*** to (effectively) translate generic code into pre-generic code by:
  - Replacing every use of a formal type parameter by a use of the most general type it could be in context (trivially, `Object`)
- This means that code compiled with Java 5 can be run by a Java 1.4 Virtual machine – there's no change to the Java bytecode.

Generics - What will be the value of res?

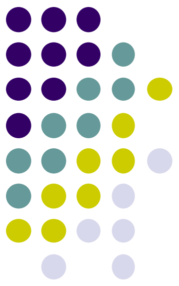


```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
Boolean res = (l1.getClass() == l2.getClass());
```

Answer: true

Explanation: after erasure both l1 ans l2 have the run-time type ArrayList

Generics - Generic usage mistakes

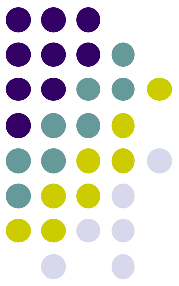


```
class MyGenClass<T, V> {
   T ob1;
   V ob2;

   // These two overloaded methods
   are ambiguous...
   void set(T o) {
      ob1 = o;
   }

   void set(V o) {
      ob2 = o;
   }
}
```

```
class Gen<T> {
   T ob;
   Gen() {
            //Can't create an instance of T...
      ob = new T();
   }
}
```

```
public class Wrong<T> {
// Wrong, no static variables of type T.
   static T ob;

// Wrong, no static method can use T.
   static T getob() {
      return ob;
   }
```

Generics - What will be printed?



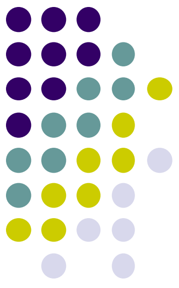
```
class Gen<T> {
  T ob;
  Gen(T o) {  ob = o;  }

  T showType() {
     println("Type of T is "+ob.getClass().getName() );
     for (Method meth : this.getClass().getDeclaredMethods())
          println(meth.toString());
      return ob;
  }

  public static void main(String args[]) {
    Gen<Integer> iOb = new Gen<Integer>(88);
    //String s = iOb.showType(); //compilation error...
    Integer i= iOb.showType();
  }
}
```

Answer:

Type of T is java.lang.Integer

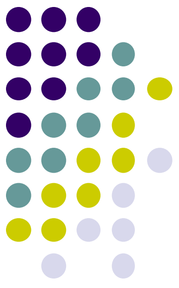
java.lang.ObjectGen.showType()

Collection Framework - General Description



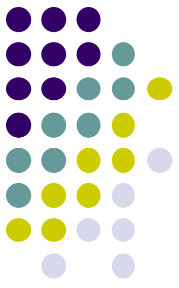
- A collection (called a container in C++) is an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve and manipulate data, and to transmit data from one method to another.
- Collections hold:
  - a specific data type;
  - a generic data type.

Collection Framework - General Description



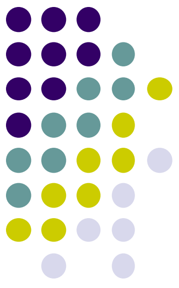
- A *collections framework* is a unified architecture for representing and manipulating collections. It has:
  - **Interfaces:** abstract data types representing collections
    - Allow collections to be manipulated independently of the details of their representation.
  - **Implementations:** concrete implementations of the collection interfaces
    - Reusable data structures
  - **Algorithms:** methods that perform useful computations, such as searching and sorting
    - These algorithms are said to be *polymorphic*: the same method can be used on different implementations.
    - Reusable functionality

Collection Framework - Why Use It?



- There are many benefits to using the Java Collections Framework:
  - Reduces programming effort.
  - Increases program speed and quality.
  - Allows interoperability among unrelated APIs.
  - Reduces the effort to learn and use new APIs.
  - Reduces effort to design new APIs.
  - Fosters software reuse.

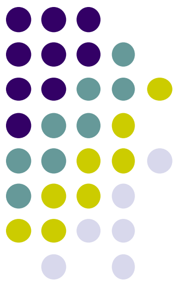
Collection Framework - Interfaces



- An interface describes a set of methods:
  - no constructors or instance variables
- Interfaces must be implemented by classes
- 2 or more classes implement an interface
  - Classes guaranteed to have the same methods
  - Objects can be treated as the same type
  - Can use different algorithms / instance variables

- Collection is actually an interface

Collection Framework - Collections Interfaces



Collection

Map

Set

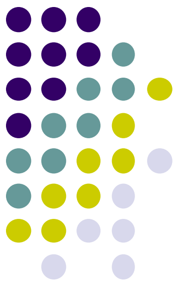
List

Queue
since Java 5

SortedMap

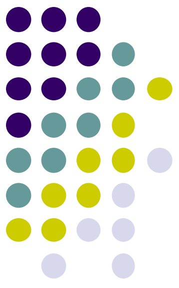
SortedSet

Collection Framework - Collection Interfaces



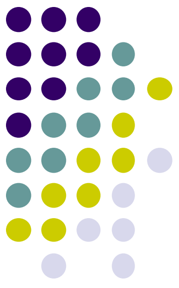
- Collection - a group of objects, called elements
  - Set - an unordered collection with no duplicates
    - SortedSet - an ordered collection with no duplicates
  - List - an ordered collection, duplicates are allowed
  - Queue -linear sequence of items "for processing"
    - Can add an item to the queue
    - Can "get the next item"from the queue
    - What is "next"depends on queue implementation
- Map - a collection that maps keys to values
  - SortedMap - a collection ordered by the keys
- Note
  - Some collections requires elements to be comparable
    - Must be able to say an element is "less than"or "greater than"another element
  - There are are two distinct hierarchies
  - We can use generics!

Collection Framework - Algorithms



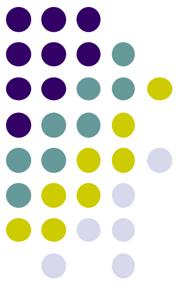
- Java has polymorphic algorithms to provide functionality for different types of collections
  - Sorting (e.g. sort)
  - Shuffling (e.g. shuffle)
  - Routine Data Manipulation (e.g. reverse, addAll)
  - Searching (e.g. binarySearch)
  - Composition (e.g. frequency)
  - Finding Extreme Values (e.g. max)

Collection Framework - Implementation (1)



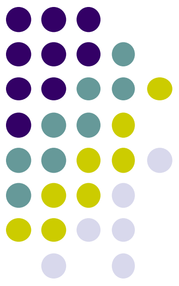
- Multiple implementations of each interface
  - All provide same basic functionality
  - Different storage requirements
  - Different performance characteristics
  - Sometimes other enhancements too
    - e.g. additional operations not part of the interface
- Java API Documentation gives the details!
  - See interface API Docs for list of implementers
  - Read API Docs of implementations for performance and storage details

Collection Framework - Implementation (2)



- A collection class
  - implements an ADT as a Java class
  - can be instantiated
  - Java implements interfaces with
    - List: ArrayList, LinkedList, *Vector, Stack…*
    - Map: HashMap, TreeMap…
    - Set: TreeSet, HashSet…
    - Queue: PriorityQueue
- All Collection implementations should have two constructors:
  - A no-argument constructor to create an empty collection
  - A constructor with another Collection as argument
  - If you implement your own Collection type, this rule cannot be enforced, because an Interface cannot specify constructors

Collection Framework -Collections and Java 1.5 Generics



- Up to Java 1.4, collections only stored Objects

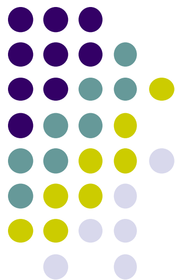
```
LinkedList points = new LinkedList();
points.add(new Point(3, 5));
Point p = (Point) points.get(0);
```

  - Casting everything gets annoying.
  - Could add non-Point objects to points collection too!

- Java 1.5 introduces generics

```
LinkedList<Point> points = new LinkedList<Point>();
points.add(new Point(3, 5));
Point p = points.get(0);
```

  - No more need for casting.
  - Can only add Point objects to points too.
  - Type checking at a compile time.



Iterable<E>

Iterator<E>

Collection<E>

ListIerator<E>

Set<E>

Queue<E>

List<E>

SortedSet<E>

EnumSet<E>

PriorityQueue<E>

ArrayList<E>

HashSet<E>

LinkedList<E>

TreeSet<E>

LinkedHashSet<E>

Map<K,V>

EnumMap<K,V>

WeakHashMap<K,V>

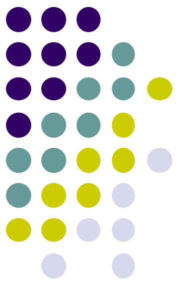
SortedMap<K,V>

HashMap<E>

TreeMap<K,V>

LinkedHashMap<K,V>

Collection Framework - The Collection Interface



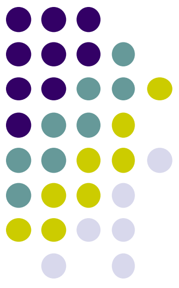
- **The Collection Interface**
  - The basis of much of the collection system is the Collection interface.

- **Methods:**
  - *public int size()*
  - *public boolean isEmpty()*
  - *public boolean contains(Object elem)*
  - *public Iterator<E> iterator()*
  - *public Object[] toArray()*
  - *public <T> T[] toArray(T[] dest)*
  - *public boolean add(E elem)*
  - *public boolean remove(Object elem)*

```
String[] strings = new
    String[collection.size()];
strings =
    collection.toArray(strings);

String[] strings =
    collection.toArray(new
    String[0]);
```

- *public boolean containsAll(Collection<?> coll)*
- *public boolean addAll(Collection<? extends E> coll)*
- *public boolean removeAll(Collection<?> coll)*
- *public boolean retainAll(Collection<?> coll)*
- *public void clear()*

Collection Framework - Collection Classes

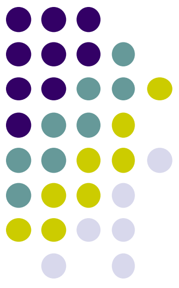


- **Classes in <u>Sets</u>:**
  - *HashSet<T>*
  - *LinkedHashSet<T>*
  - *TreeSet<T>*
  - *EnumSet<T extends Enum<T>>*

- **Classes in <u>Lists</u>:**
  - *To define a collection whose elements have a defined order-each element exists in a praticular poistion the collection.*
  - *Vector<T>*
  - *Stack<T>*
  - *LinkedList<T>*
  - *ArrayList<T>*

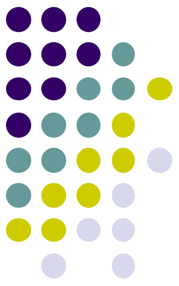
- **Class in <u>Queues</u>:**
  - *FIFO ordering*
  - *PriorityQueue<T>*
- **Classes in <u>Maps</u>:**
  - *Does not extend Collection because it has a contract that is different in important ways: do not add an element to a Map(add a key/value pair), and a Map allows looking up.*
  - *Hashtable<K,V>*
  - *HashMap<K,V>*
  - *LinkedHashMap<K,V>*
  - *WeakHashMap<K,V>*
  - *IdentityHashMap<K,V>*
  - *TreeMap<K,V> : keeping its keys sorted in the same way as TreeSet*

Collection Framework - Collections of Objects (1)



- Sequences
  - The objects are stored in a linear fashion, not necessarily in any particular order, but in an arbitrary fixed sequence with a beginning and an end.
  - Collections generally have the capability to expand to accommodate as many elements as necessary.
- Maps
  - Each entry in the collection involves a pair of objects.
  - A map is also referred to sometimes as a dictionary.
  - Each object that is stored in a map has an associated key object, and the object and its key are stored together as a “name-value” pair.

Collection Framework - Using Collections



- Lists and sets are easy:

```
HashSet<String> wordList = new HashSet<String>();
LinkedList<Point> waypoints = new LinkedList<Point>();
```

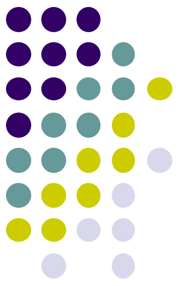
  - Element type must appear in both variable declaration and in new-expression.

- Maps are more verbose:

```
TreeMap<String, WordDefinition> dictionary =
    new TreeMap<String, WordDefinition>();
```

  - First type is key type, second is the value type.
- See Java API Docs for available operations

Collection Framework - Iteration Over Collections



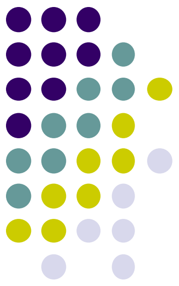
- Often want to iterate over values in collection.

  ArrayList collections are easy:

```
ArrayList<String> quotes;
…
for (int i = 0; i < quotes.size(); i++)
      System.out.println(quotes.get(i));
```

  - Impossible/undesirable for other collections!
    - Iteratorsare used to traverse contents
- Iterator is another simple interface:
  - hasNext() –Returns true if can call next()
  - next() –Returns next element in the collection
- ListIterator extends Iterator
  - Provides many additional features over Iterator

Collection Framework - Iteration Over Collections (2)



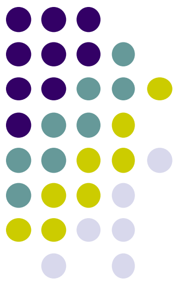
- Collections provide an **iterator()** method
  - Returns an iterator for traversing the collection

- Example:

```
HashSet<Player> players;
...
Iterator<Player> iter = players.iterator();
    while (iter.hasNext()) {
    Player p = iter.next();
    ... // Do something with p
}
```

- Iterator should also use generics
- Can use iterator to delete current element, etc.

Collection Framework -Java 1.5 Enhanced For-Loop Syntax



- Setting up and using an iterator is annoying
- Java 1.5 introduces syntactic sugar for this:

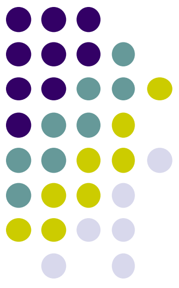
```
for (Player p : players) {
    … // Do something with p
}
```

  - Can't access actual iterator used in loop.
  - Best for simple scans over a collection's contents

- Can also use enhanced for-loop syntax with arrays:

```
float sum(float[] values) {
    float result = 0.0f;
    for (float val : values) result += val;
    return result;
}
```

Collection Framework - Iterators and ListIterators



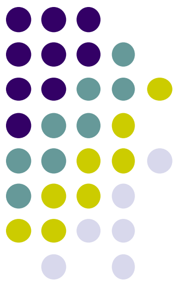
- **Iterator<E> interface**
  - *T next()*
  - *boolean hasNext()*
  - *void remove()*

```
public void removeLongStrings
 (Collection<? Extends String> coll, int maxLen) {
 Iterator<? Extends String> it = coll.iterator();
 while (it.hasNext()) {
  String str = it.next();
  if (Str.length() > maxLen) it.remove();
 }
}
```

- **ListIterator<E> interface**
  - *extends Iterator*
  - *T next()*
  - *boolean hasNext()*
  - *int nextIndex()*
  - *T previous()*
  - *boolean hasPrevious()*
  - *int previousIndex()*
  - *void remove()*
  - *void add(T obj)*
  - *void set(T obj)*

```
ListIterator<String> it = list.listIterator(list.size());
while (it.hasPrevious()) {
  String obj = it.previous();
  System.out.println(obj);
  // ... use obj  ....
}
```

Collection Framework - Collection Algorithms



- java.util.Collections class provides some common algorithms
  - …not to be confused with Collection interface
  - Algorithms are provided as static functions.
  - Implementations are fast, efficient, and generic.

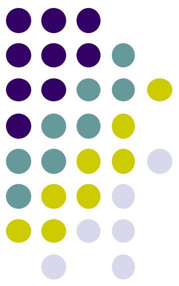
- Example: sorting

```
LinkedList<Product> groceries;
…
Collections.sort(groceries);
```

  - Collection is sorted in-place: groceriesis changed

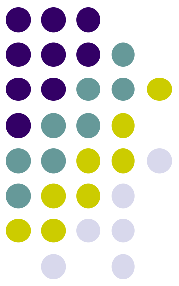
- Read Java API Docs for more details
- Also see Arrays class for array algorithms

Collection Framework - Collection Elements (1)



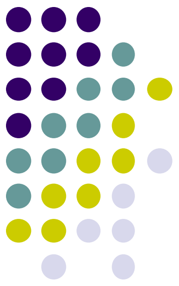
- Collection elements may require certain capabilities.
- List elements don't need anything special
  - ...unless contains(), remove(), etc. are used!
  - Then, elements should provide a correct equals() implementation
- Requirements for equals():
  - a.equals(a) returns true
  - a.equals(b) same as b.equals(a)
  - If a.equals(b)is true and b.equals(c)is true, then a.equals(c)is also true
  - a.equals(null)returns false

Collection Framework - Collection Elements (2)



- Sets and maps require special features
  - Sets require these operations on set-elements
  - Maps require these operations on the keys
- equals() must definitely work correctly
- TreeSet, TreeMap require sorting capability
  - Element or key class must implement java.lang.Comparable interface
  - Or, an appropriate implementation of java.util.Comparator must be provided
- HashSet, HashMap require hashing capability
  - Element or key class must provide a good implementation of Object.hashCode()

Collection Framework - Implementing hashCode(1)



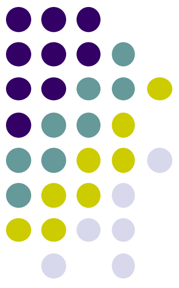
- Is this a correct implementation?

```
public int hashCode() {
        return 42;
}
```

  - It satisfies the rules, so technically yes…
  - In practice, will cause programs to be very inefficient.

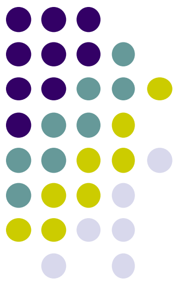
- Hash function should generate a wide range of values.
  - Specifically, should produce a uniform distribution of values.
  - Facilitates most efficient operation of hash tables.
  - Requirement is that equal objects must produce identical hash values…
  - Also good if unequal objects produce different hash values.

Collection Framework - Implementing hashCode(2)



- A few basic hints:
  - If field is a boolean, use 0 or 1 for hash code
  - If field is an integer type, cast value to int
  - If field is a non-array object type:
    - Call the object's hashCode() function, or use 0 for null
  - If field is an array:
    - Include every array-element into final hash value!
- If computing the hash is expensive, cache it.
  - Must re-compute hash value if object changes!

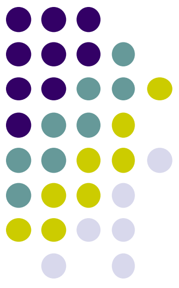
Collection Framework -Comparing and Ordering Objects



- Objects implement **java.lang.Comparable<T>interface to allow them to be ordered**
  - **public int compareTo(T obj)**
- Returns a value that imposes an order:
  - result < 0 means **thisis less than obj**
  - result == 0 means **thisis “same as”obj**
  - result > 0 means **thisis greater than obj**
- This defines the *natural orderingof a class*
  - *i.e. the “usual”or “most reasonable”sort-order*
- Natural ordering should be *consistent with*
  - ***equals()***
  - ***a.compareTo(b)returns 0 only when a.equals(b)is true***

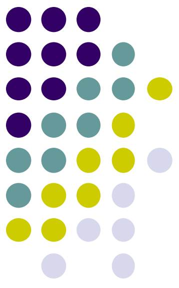
- Implement this interface correctly for using **TreeSet/ TreeMap**

Collection Framework - Alternate Orderings



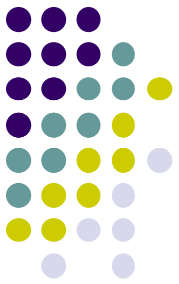
- Can provide extra comparison functions.
  - Provide a separate object that implements java.util.Comparator<T> interface
  - Simple interface:
    - int compare(T o1, T o2)
- Sorted collections, sort algorithms can also take a comparator object.
  - Allows sorting by all kinds of things!
- Comparator implementations are typically nested classes
  - e.g. Playerclass could provide a ScoreComparator nested class

Reflection - Java looking at Java



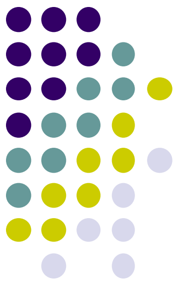
- One of the unusual capabilities of Java is that a program can examine itself
  - You can determine the class of an object
  - You can find out all about a class: its access modifiers, superclass, fields, constructors, and methods
  - You can find out what is in an interface
  - Even if you don't know the names of things when you write the program, you can:
    - Create an instance of a class
    - Get and set instance variables
    - Invoke a method on an object
    - Create and manipulate arrays
- In "normal" programs you don't need reflection
- You *do* need reflection if you are working with programs that process programs
  - Debugger

Reflection - Introspection



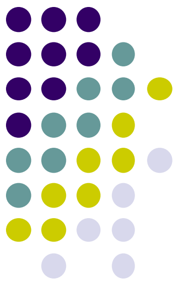
- Introspection is a programmatic facility built on top of reflection and a few supplemental specifications (see the java.beans package).
- It provides somewhat higher-level information about a class than does reflection.
- *Introspection* makes general class information available at run-time
  - The type (class) does not have to be known at compile time
  - E.g. list the attributes of an object
- This is very useful in
  - Rapid Application Development (RAD)
    - Visual approach to GUI development
    - Requires information about component at run-time
      - JavaBeans
  - Remote Method Invocation (RMI)
    - Distributed objects

Reflection - The Class class



- To find out about a class, first get its Class object
  - If you have an object obj, you can get its class object with
    Class c = obj.getClass();
  - You can get the class object for the superclass of a Class c with
    Class sup = c.getSuperclass();
  - If you know the name of a class (say, Button) at compile time, you can get its class object with
    Class c = Button.class;
  - If you know the name of a class at run time (in a String variable str), you can get its class object with
    Class c = class.forName(str);

Reflection - Getting the class name



- If you have a class object c, you can get the name of the class with c.getName()
- getName returns the fully qualified name; that is,

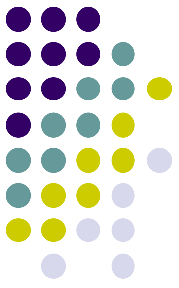
```
Class c = Button.class;
String s = c.getName();
System.out.println(s);
```

  will print

```
java.awt.Button
```

- Class Class and its methods are in java.lang, which is always imported and available

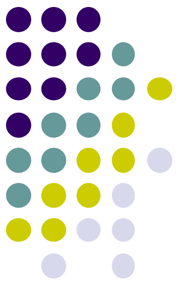
Reflection - Getting all the superclasses



- getSuperclass() returns a Class object (or null if you call it on Object, which has no superclass)
- The following code is from the Sun tutorial:

```
static void printSuperclasses(Object o) {
    Class subclass = o.getClass();
    Class superclass = subclass.getSuperclass();
    while (superclass != null) {
        String className = superclass.getName();
        System.out.println(className);
        subclass = superclass;
        superclass = subclass.getSuperclass();
    }
}
```

Reflection - Getting the class modifiers(1)



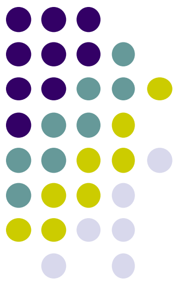
- The modifiers (e.g., public, final, abstract etc.) of a Class object is encoded in an int and can be queried by the method getModifiers().
- To decode the int result, we need methods of the Modifier class, which is in java.lang.reflect, so:

```
import java.lang.reflect.*;
```

- Then we can do things like:

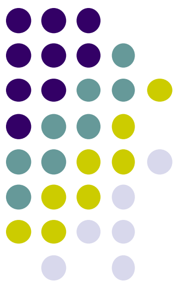
```
if (Modifier.isPublic(m))
    System.out.println("public");
```

Reflection - Getting the class modifiers (2)



- Modifier contains these methods (among others):
  - public static boolean isAbstract(*int*)
  - public static boolean isFinal(*int*)
  - public static boolean isInterface(*int*)
  - public static boolean isPrivate(*int*)
  - public static boolean isProtected(*int*)
  - public static boolean isPublic(*int*)
  - public static String toString(*int*)
    - This will return a string such as "public final synchronized strictfp"

Reflection - Getting interfaces

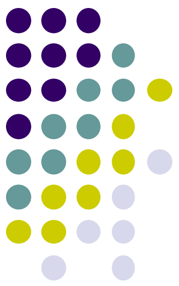


- A class can implement zero or more interfaces
- getInterfaces() returns an *array* of Class objects

```
static void printInterfaceNames(Object o) {
    Class c = o.getClass();
    Class[] theInterfaces = c.getInterfaces();
    for (Class inf: interfaces) {
        System.out.println(inf.getName());    }}
```

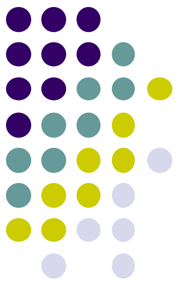
- The class Class represents both classes and interfaces
- To determine if a given Class object *c* is an interface, use c.isInterface()
- To find out more about a class object, use:
  - getModifiers(), getFields() // "fields" == "instance variables“, getConstructors(), getMethods(), isArray()

Reflection - Getting Fields



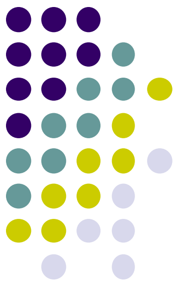
- public Field[] getFields() throws SecurityException
  - Returns an array of *public* Fields (including inherited fields).
  - The length of the array may be zero
  - The fields are not returned in any particular order
  - Both locally defined and inherited instance variables are returned, but *not* static variables.
- public Field getField(String name)
  throws NoSuchFieldException, SecurityException
  - Returns the named *public* Field
  - If no immediate field is found, the superclasses and interfaces are searched recursively

Reflection - Using Fields



- If *f* is a Field object, then
  - *f*.getName() returns the simple name of the field
  - *f*.getType() returns the type (Class) of the field
  - *f*.getModifiers() returns the Modifiers of the field
  - *f*.toString() returns a String containing access modifiers, the type, and the fully qualified field name
    - Example: public java.lang.String Person.name
  - *f*.getDeclaringClass() returns the Class in which this field is declared
    - note: getFields() may return superclass fields.

Reflection - Getting Constructors of a class



- if c is a Class, then
- c.getConstructors() : Constructor[] return an array of all public constructors of class c.
- c.getConstructor(Class … paramTypes) returns a constructor whose parameter types match those given paramTypes.

Ex:

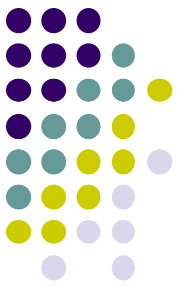
- String.class.getConstructors().length

> 15;

- String.class.getConstrucor(char[].class, int.class, int.class).toString()

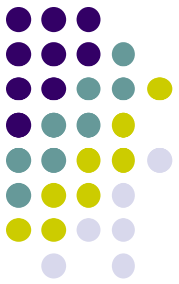
> String(char[], int,int).

Reflection - Constructors



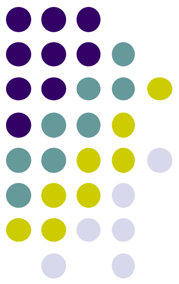
- If *c* is a Constructor object, then
  - c.getName() returns the name of the constructor, as a String (this is the same as the name of the class)
  - c.getDeclaringClass() returns the Class in which this constructor is declared
  - c.getModifiers() returns the Modifiers of the constructor
  - c.getParameterTypes() returns an array of Class objects, in declaration order
  - c.newInstance(Object… initargs) creates and returns a new instance of class *c*
    - Arguments that should be primitives are automatically unwrapped as needed

Reflection - Example



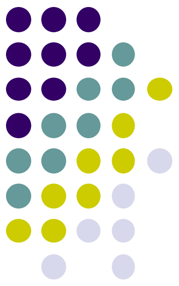
- Constructor c = String.class.getConstrucor(char[].class, int.class, int.class).toString()
- String(char[], int,int).
- String s = c.newInstance(
                    new char[] {‘a’,’b’,’c’,’d’ }, 1, 2);
- assert s == “bc”;

Reflection - Methods



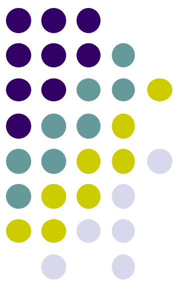
- public Method[] getMethods()
  throws SecurityException
  - Returns an array of Method objects
  - These are the *public member* methods of the class or interface, including inherited methods
  - The methods are returned in no particular order
- public Method getMethod(String name,
  Class… parameterTypes)
  throws NoSuchMethodException, SecurityException

Reflection - Method methods (1)



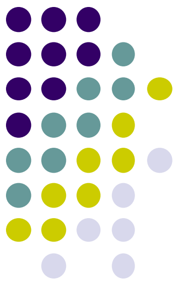
- getDeclaringClass()
  - Returns the Class object representing the class or interface that declares the method represented by this Method object
- getName()
  - Returns the name of the method represented by this Method object, as a String
- getModifiers()
  - Returns the Java language modifiers for the method represented by this Method object, as an integer
- getParameterTypes()
  - Returns an array of Class objects that represent the formal parameter types, in declaration order, of the method represented by this Method object

Reflection - Method methods (2)



- getReturnType()
  - Returns a Class object that represents the formal return type of the method represented by this Method object
- toString()
  - Returns a String describing this Method (typically pretty long)
- public Object invoke(Object obj, Object... args)
  - Invokes the underlying method represented by this Method object, on the specified object with the specified parameters
  - Individual parameters are automatically unwrapped to match primitive formal parameters

Reflection - Examples of invoke()



- “abcdefg”.length()

> 7

- Method lengthMethod = String.class.getMethod(“length”) ;
- lengthMethod.invoke(“abcdefg”)

> 7

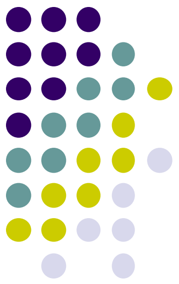
- “abcdefg”.substring(2, 5)

> cde

- Method substringMethod = String.class.getMethod (“substring”, int.class, Integer.TYPE) ;
- substringEMthod.invoke(“abcdefg”, 2, new Integer(5))

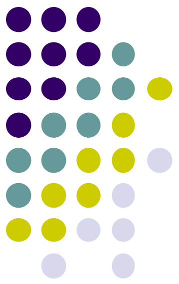
> cde

Reflection - Arrays (1)



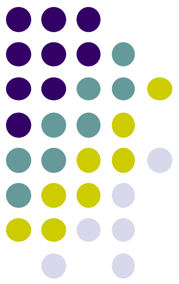
- To determine whether an object obj is an array,
  - Get its class c with Class c = obj.getClass();
  - Test with c.isArray()
- To find the type of components of the array,
  - c.getComponentType()
    - Returns null if c is not the class of an array
- Ex:
  - int[].class.isArray() == true ;
  - int[].class.getComponentType() == int.class

Reflection - Arrays (2)



- The Array class in java.lang.reflect provides *static* methods for working with arrays
- To create an array,
- Array.newInstance(Class componentType, int size)
  - This returns, as an Object, the newly created array
    - You can cast it to the desired type if you like
  - The componentType may itself be an array
    - This would create a multiple-dimensioned array
    - The limit on the number of dimensions is usually 255
- Array.newInstance(Class componentType, int... sizes)
  - This returns, as an Object, the newly created multidimensional array (with sizes.length dimensions)

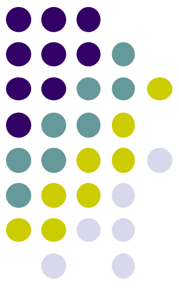
Reflection - Examples



- a = new int[] {1,2,3,4};
- Array.getInt(a, 2) // → 3
- Array.setInt(a, 3, 5) // a = {1,2,3, 5 }.

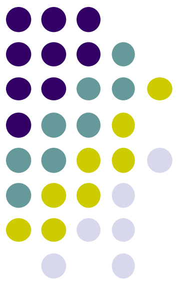
- s = new String[] { “ab”, “bc”, “cd” };
- Array.get(s, 1) // → “bc”
- Array.set(s, 1, “xxx”) // s[1] = “xxx”

Reflection - Getting non-public members of a class



- All getXXX() methods of Class mentioned above return only public members of the target (as well as ancestor) classes, but they cannot return non-public members.
- There are another set of getDeclaredXXX() methods in Class that will return all (even private or static) members of target class but no inherited members are included.
- getDeclaredConstructors(), defDeclaredConstrucor(Class…)
- getDeclaredFields(),
  getDeclaredField(String)
- getDeclaredmethods(),
  getDeclaredMethod(String, Class…)

Reflection - Example



- String.class.getConstructors().length

> 15

- String.class.getDeclaredConstructors().length

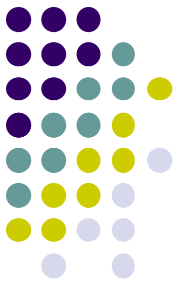
> 16.

- Constructor[] cs = String.class.getDeclaredConstructors();

```
for(Constructor c : cs)
   if( ! (Modifier.isPublic(c.getModifiers())))
      out.println(c);
```

> java.lang.String(int,int,char[]) // package

Annotations - History

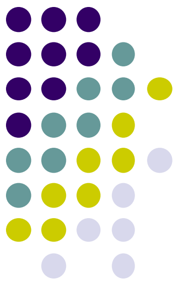


- The Java platform has always had various ad hoc annotation mechanisms
  - Javadoc annotations

```
/**
* Locate a value in a
* collection.
* @param value the sought-after value
* @return the index location of the value
* @throws NotFoundException
*/
int search( Object value ) { …
```

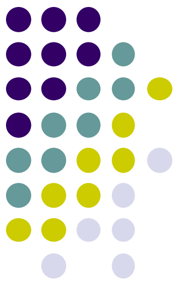
  - **`@transient`** – an ad hoc annotation indicating that a field should be ignored by the serialization subsystem
  - **`@deprecated`** – an ad hoc annotation indicating that the method should no longer be used

Annotations - Introduction



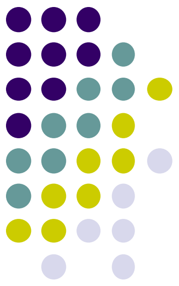
- Annotations provide data about a program that is not part of the program itself. An **annotation** is an attribute of a program element.
- As of release 5.0, the platform has a general purpose annotation (metadata) facility that permits to define and use **your own** annotation types.
- The facility consists of:
  - a syntax for declaring annotation types
  - a syntax for annotating declarations
  - APIs for reading annotations
  - a class file representation for annotations
  - an annotation processing tool

Annotations - Usage



- Annotations have a number of uses, among them:
  - **Information for the compiler** - Annotations can be used by the compiler to detect errors or suppress warnings
  - **Compiler-time and deployment-time processing** - Software tools can process annotation information to generate code, XML files, and so forth
  - **Runtime processing** - Some annotations are available to be examined at runtime (reflection)

Annotations - Annotation Type Declaration (1)

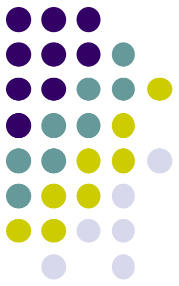


- Similar to normal interface declarations:

```
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date();    default "[unimplemented]";
}
```

- An at-sign @ precedes the interface keyword
- Each method declaration defines an element of the annotation type
- Methods can have default values
- Once an annotation type is defined, you can use it to annotate declarations

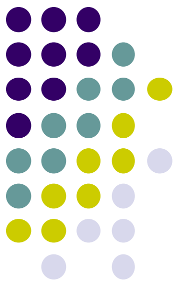
Annotations - Annotation Type Declaration (2)



- Method declarations should not have any parameters
- Method declarations should not have any throws clauses
- Return types of the method should be one of the following:
  - primitives, String, Class, enum, array of the above types

```
public @interface RequestForEnhancement {
    int    id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date();    default "[unimplemented]";
}
```

Annotations - Annotating Declarations (1)

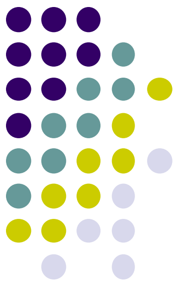


- Syntactically, the annotation is placed in front of the program element's declaration, similar to **`static`** or **`final`** or **`protected`**

```
@RequestForEnhancement(
    id        = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date      = "4/1/3007"
)
public static void travelThroughTime(Date destination) { ... }
```

- An annotation instance consists of
  - the "@" sign
  - the annotation name
  - a parenthesized list of name-value pairs

Annotations - Annotating Declarations (2)



- In annotations with a single element, the element should be named **`value`**:

```
public @interface Copyright {
    String value();
}
```

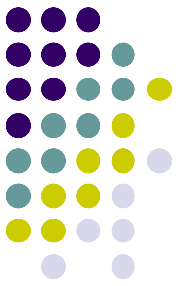
- It is permissible to omit the element name and equals sign (=) in a single-element annotation:

```
@Copyright("2002 Yoyodyne Propulsion Systems")
public class OscillationOverthruster { ... }
```

- If no values, then no parentheses needed:

```
public @interface Preliminary { }

@Preliminary public class TimeTravel { ... }
```

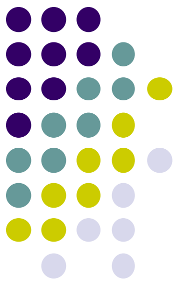
Annotations - What can be annotated?



Annotatable program elements:

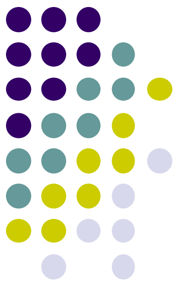
- package
- class, including
  - interface
  - enum
- method
- field
- only at compile time
  - local variable
  - formal parameter

Annotations - Annotations Used by the Compiler



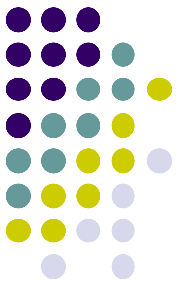
- There are three annotation types that are predefined by the language specification itself:
  - `@Deprecated` – indicates that the marked element is deprecated and should no longer be used
  - `@Override` – informs the compiler that the element is meant to override an element declared in a superclass
  - `@SuppressWarnings` – tells the compiler to suppress specific warnings that it would otherwise generate

Annotations - Meta-Annotations



- **Meta-annotations** - types designed for annotating annotation-type declarations (annotations-of-annotations)
- Meta-annotations:
  - `@Target` - indicates the targeted elements of a class in which the annotation type will be applicable
    - TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, etc
  - `@Retention` - how long the element holds onto its annotation
    - SOURCE, CLASS, RUNTIME
  - `@Documented` - indicates that an annotation with this type should be documented by the javadoc tool
  - `@Inherited` - indicates that the annotated class with this type is automatically inherited

Annotations - Annotation Processing

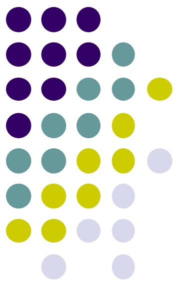


- It's possible to read a Java program and take actions based on its annotations
- To make annotation information available at runtime, the annotation type itself must be annotated with **`@Retention(RetentionPolicy.RUNTIME)`**:

```
@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationForRuntime
{
   // Elements that give information for runtime processing
}
```

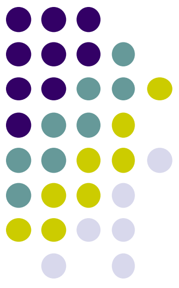
- Annotation data can be examined using reflection mechanism, see e.g. **`java.lang.reflect.AccessibleObject`**:
  - `<T extends Annotation> T getAnnotation(Class<T>)`
  - `Annotation[] getAnnotations()`
  - `boolean isAnnotationsPresent(<Class<? extends Annotation>)`

Annotations - Bigger Example

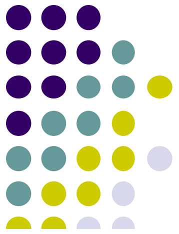


- The following example shows a program that pokes at classes to see "if they illustrate anything"
- Things to note in example:
  - An annotation may be annotated with itself
  - How annotations meta-annotated with **`Retention(RUNTIME)`** can be accessed via reflection mechanisms

Annotations - Class Annotation Example

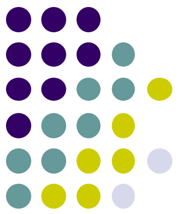


```
@Retention(value=RetentionPolicy.RUNTIME)
@Illustrate( {
   Illustrate.Feature.annotation,
   Illustrate.Feature.enumeration } )
public @interface Illustrate {
    enum Feature {
        annotation, enumeration, forLoop,
        generics, autoboxing, varargs;

        @Override public String toString() {
            return "the " + name() + " feature";
        }
    };
    Feature[] value() default {Feature.annotation};
}
```



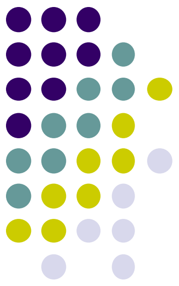
```
import java.lang.annotation.Annotation;

@Author(@Name(first="James",last="Heliotis"))
@Illustrate(
    {Illustrate.Feature.enumeration,Illustrate.Feature.forLoop})
public class Suggester {
    @SuppressWarnings({"unchecked"}) // not yet supported
    public static void main( String[] args ) {
        try {
            java.util.Scanner userInput =
                                new java.util.Scanner( System.in );
            System.out.print( "In what class are you interested? " );
            Class theClass = Class.forName( userInput.next() );
            Illustrate ill =
              (Illustrate)theClass.getAnnotation( Illustrate.class );
```

… continued …



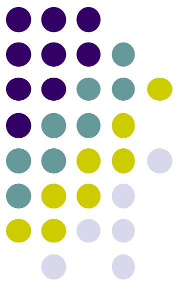
```
            if ( ill != null ) {
                System.out.println( "Look at this class if you'd " +
                                    " like to see examples of" );
                for ( Illustrate.Feature f : ill.value() ) {
                    System.out.println( "\t" + f );
                }
            }
            else {
                System.out.println(
                          "That class will teach you nothing." );
            }
        }
        catch( ClassNotFoundException cnfe ) {
            System.err.println( "I could not find a class named \"" +
                                cnfe.getMessage() + "\"." );
            System.err.println( "Are you sure about that name?" );
        }
    }
}
```

Annotations - Compilation and Execution



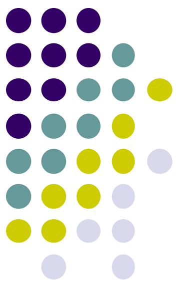
```
$ javac *.java
Note: Suggester.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

$ java Suggester
In what class are you interested? Suggester
Look at this class if you'd like to see examples of
        the enumeration feature
        the forLoop feature

$ java Suggester
In what class are you interested? Illustrate
Look at this class if you'd like to see examples of
        the annotation feature
        the enumeration feature
```

Annotations - Execution



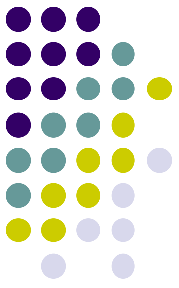
```
$ java Suggester
In what class are you interested? Coin
That class will teach you nothing.

$ java Suggester
In what class are you interested? Foo
I could not find a class named "Foo".
Are you sure about that name?
```

Annotations - Example - JPA Annotations



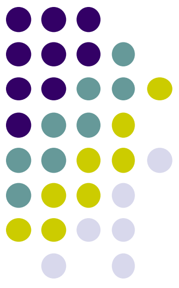
- When using JPA, you can configure the JPA behavior of your entities using annotations:
  - `@Entity` - designate a plain old Java object (POJO) class as an entity so that you can use it with JPA services
  - `@Table`, `@Column`, `@JoinColumn`, `@PrimaryKeyJoinColumn` – database schema attributes
  - `@OneToOne`, `@ManyToMany` – relationship mappings
  - `@Inheritance`, `@DiscriminatorColumn` – inheritance controlling

Annotations - Example - JUnit Annotations



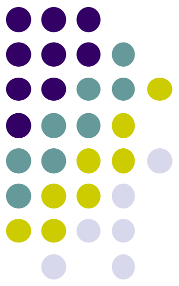
- Annotations and support for Java 5 are key new features of JUnit 4:
  - `@Test` – annotates test method
  - `@Before, @After`– annotates setUp() and tearDown() methods for each test
  - `@BeforeClass, @AfterClass` – class-scoped setUp() and tearDown()
  - `@Ignore` – do not run test

Java I/O – Reading & Writing Data



- Data can come from many Sources & go to many Destinations
  - Memory
  - Disk
  - Network
- Whatever the Source or Destination, a Stream has to be opened to Read/Write Data

Java I/O – Reading & Writing Data



Reading

Open a Stream

While more Information

Read

Close the Stream

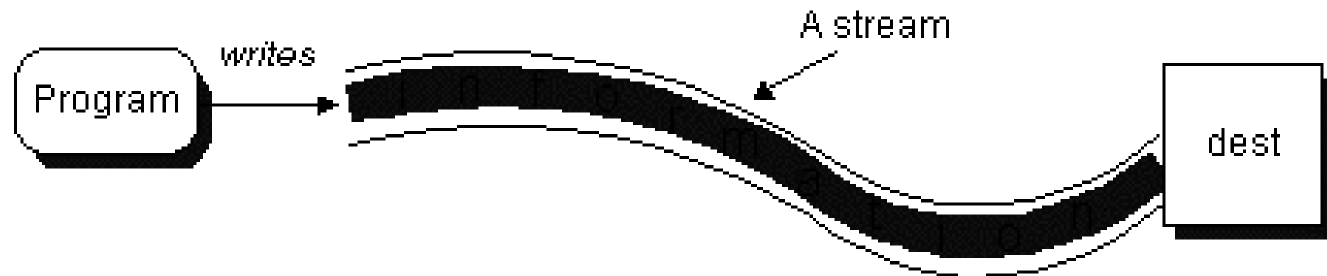
Writing

Open a Stream

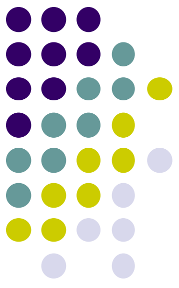
While more Information

Write

Close the Stream

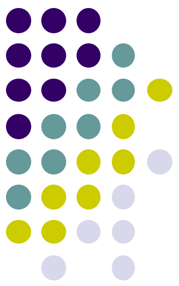


Java I/O - Reading & Writing Data



- java.io Package includes these Stream Classes
  - *Character Streams* are used for 16-bit Characters – Uses *Reader* & *Writer* Classes
  - *Byte Streams* are used for 8-bit Bytes – Uses *InputStream* & *OutputStream* Classes Used for Image, Sound Data etc.

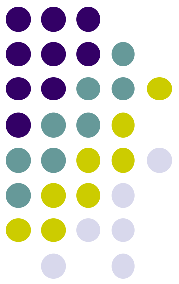
Java I/O - Character Streams



- Reader and Writer are abstract super classes for character streams (16-bit data)
- Sub classes provide specialized behavior

Reader
- BufferedReader
  - LineNumberReader
- CharArrayReader
- InputStreamReader
  - FileReader
- FilterReader
  - PushbackReader
- PipedReader
- StringReader

Writer
- BufferedWriter
- CharArrayWriter
- OuputStreamReader
  - FileWriter
- FilterWriter
- PipedWriter
- StringWriter
- FilterWriter



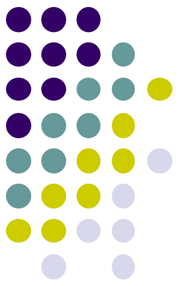
Java I/O - Byte Streams

- InputStream and OutoutStream are abstract super classes for byte streams (8-bit data)
- Sub classes provide specialized behavior

- InputStream
  - FileInputStream
  - PipedInputStream
  - FilterInputStream
    - LineNumberInputStream
    - DataInputStream
    - BufferedInputStream
    - PushbackInputStream
  - ByteArrayInputStream
  - SequenceInputStream
  - StringBufferInputStream
  - ObjectInputStream

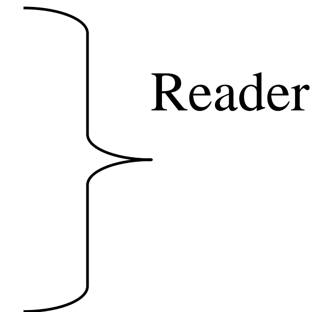
- OutputStream
  - FileOutputStream
  - PipedOutputStream
  - FilterOutputStream
    - DataOutputStream
    - BufferedOutputStream
    - PrintStream
  - ByteArrayOutputStream
  - ObjectOutputStream

Java I/O - I/O Super Classes (1)

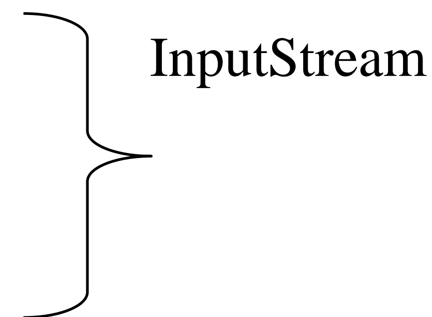


- Reader and InputStream define similar APIs but for different data types

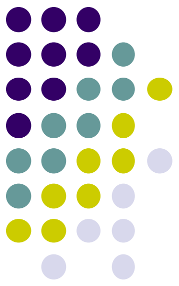
```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
```




```
int read()
int read(byte cbuf[])
int read(byte cbuf[], int offset, int length)
```



Java I/O - I/O Super Classes (2)



- Writer and OutputStream define similar APIs but for different data types

```
int write()
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

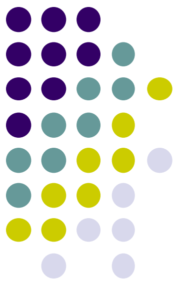
Writer

```
int write()
int write(byte cbuf[])
int write(byte cbuf[], int offset, int length)
```

OutputStream

Type of I/O	Streams	Description
Memory	CharArrayReader CharArrayWriter ByteArrayInputStream ByteArrayOutputStream	Use these streams to read from and write to memory. You create these streams on an existing array and then use the read and write methods to read from or write to the array.
	StringReader StringWriter StringBufferInputStream	Use StringReader to read characters from a String in memory. Use StringWriter to write to a String. StringWriter collects the characters written to it in a StringBuffer, which can then be converted to a String. StringBufferInputStream is similar to StringReader, except that it reads bytes from a StringBuffer.
Pipe	PipedReader PipedWriter PipedInputStream PipedOutputStream	Implement the input and output components of a pipe. Pipes are used to channel the output from one thread into the input of another.
File	FileReader FileWriter FileInputStream FileOutputStream	Collectively called file streams, these streams are used to read from or write to a file on the native file system.
Object Serializati-on	*N/A* ObjectInputStream ObjectOutputStream	Used to serialize objects.

Java I/O - Stream wrapping



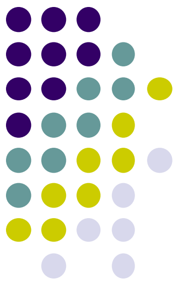
- BufferedReader class can be used for efficient reading of characters, arrays and lines

```
BufferedReader in = new BufferedReader(new FileReader("foo.in"));
```

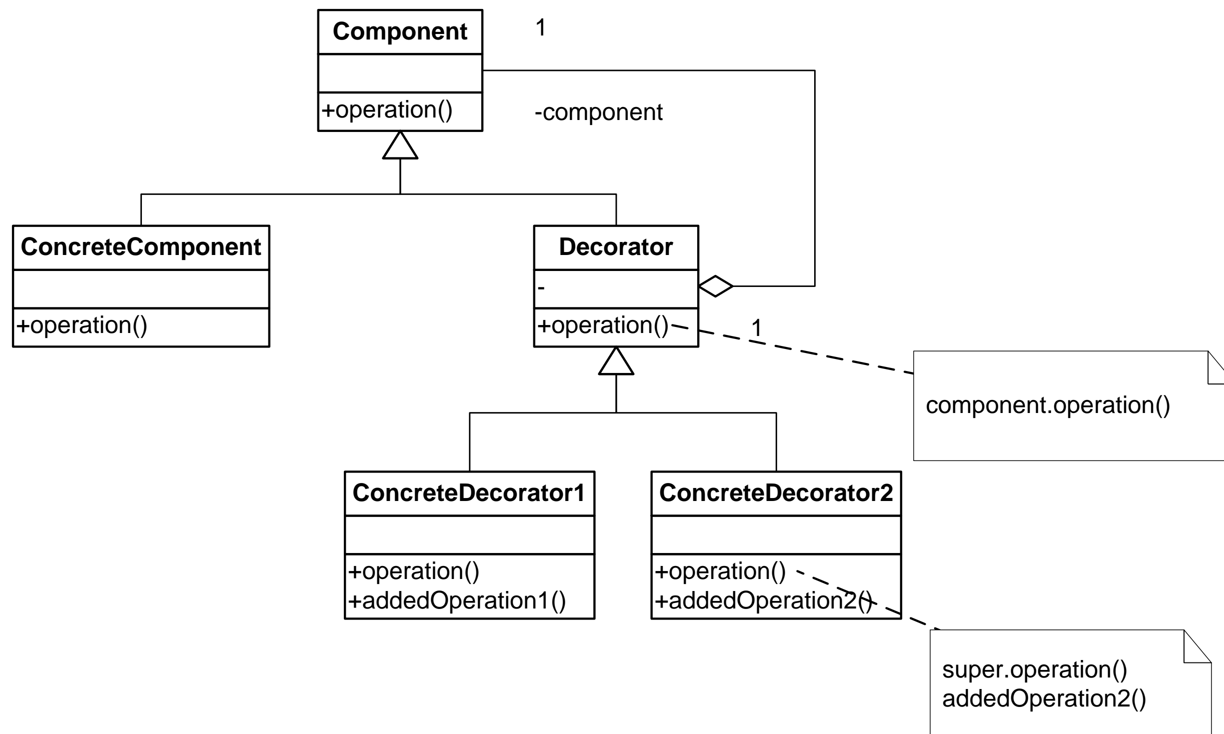
- BufferedWriter and PrintWriter classes can be used for efficient writing of characters, arrays and lines and other data types

```
BufferedWriter out = new BufferedWriter(newFileWriter("foo.out"));

PrintWriter out= new PrintWriter(new BufferedWriter(new FileWriter("foo.out")));
```

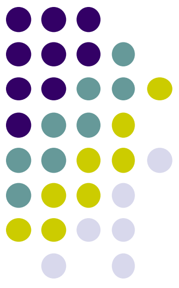
Java I/O - Decorator Pattern



- Capabilities are added using a design called the **Decorator Pattern.**

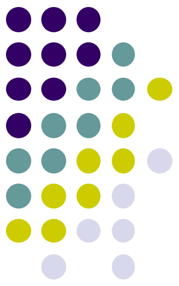


Java I/O – Purpose of Decorator



- Best way to think of this is as follows:
  - There are two important issues when constructing an i/o library
    - Where the i/o is going (file, etc).
    - How the data is represented (String, native type, etc.)
  - Rather than create a class for each combination, Decorator classes allow you to mix and match, augment functionality of base classes.
  - This is a bit confusing but is very flexible.
  - Decorators can also add other capabilities, such as peek ahead, push back, write line number, etc.

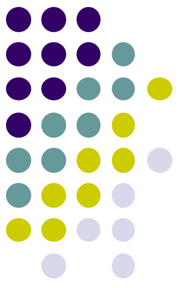
Java I/O - Java decorators



- All Java i/o decorator classes inherit from FilterInputStream and FilterOutputStream
- Look at the api for these classes and note a few things:
  - They wrap instances of InputStream/OutputStream respectively.
  - They inherit from InputStream/OutputStream respectively
- This is an odd inheritence hierarchy but is necessary to ensure that the FilterStreams support the same interface as the underlying class.

```
public class FilterInputStream extends InputStream {

        protected InputStream in;

        protected FilterInputStream(InputStream in) {

                this.in = in;

} }
```

Java I/O - File Handling - Character Streams



```
import java.io.*;
public class CopyCharacters {
public static void main(String[] args) throws IOException {
    File inputFile = new File("InputFile.txt");
    File outputFile = new File("OutputFile.txt");
    FileReader in = new FileReader(inputFile);
    FileWriter out = new FileWriter(outputFile);
    int c;

    while ((c = in.read() ) != -1)      // Read from Stream
            out.write(c);               // Write to Stream
    in.close();
    out.close();
    }
}
```

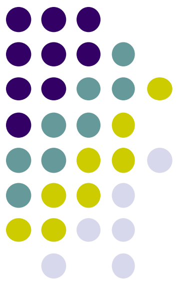
Why is that needed?

Create File Objects

Create File Streams

Close the Streams

Java I/O - Getting User Input in Command Line

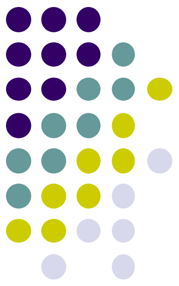


- Read as reading from the standard input device which is treated as an input stream represented by System.in

```
BufferedReader input= new
        BufferedReader(newInputStreamReader(System.in));
System.out.println("Enter the name :"  );
String name =input.readLine();
```

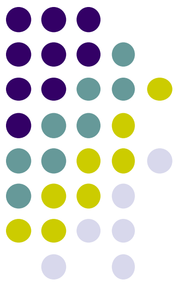
- Throws java.io.IOException

Java I/O - Object Serialization



- To allow to *Read & Write Objects*
- The State of the Object is represented in a *Serialized* form sufficient to reconstruct it later
- Streams to be used
  - *ObjectInputStream*
  - *ObjectOutputStream*

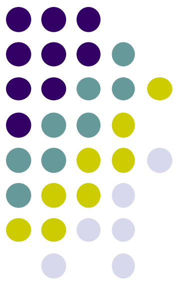
Java I/O - Object Serialization



- Object Serialization is used in
  - Remote Method Invocation (RMI) : communication between objects via sockets
  - Lightweight persistence : the archival of an object for use in a later invocation of the same program
- An Object of any Class that implements the *Serializable Interface* can be serialized
  - public class MyClass implements Serializable {

    ```
        ...
    }
    ```
- Serializable is an Empty Interface, no methods have to be implemented

Java I/O - Object Serialization Example

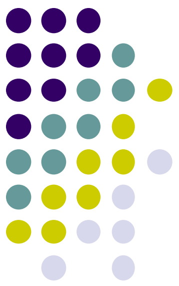


- Writing to an ObjectOutputStream

```
FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();
```

- ObjectOutputStream must be constructed on another Stream

Java I/O - Object Serialization



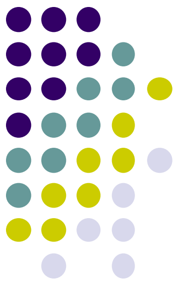
- Reading from an ObjectInputStream

```
FileInputStream in = new FileInputStream("Time");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Why is casting needed here?

- The objects must be read from the stream in the same order in which they were written

Java I/O - Object Serialization

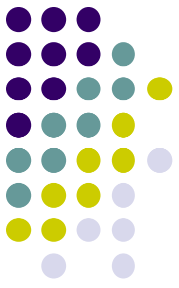


- Specialized behavior can be provided in serilazation and deserialization by implementing the following methods

```
private void writeObject(java.io.ObjectOutputStream out) throws
    IOException

private void readObject(java.io.ObjectInputStream in) throws
    IOException, ClassNotFoundException;
```

Hey, what's that!?

Java I/O - Protecting sensitive data



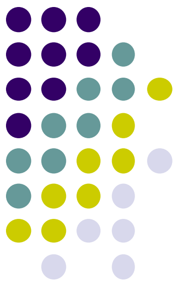
Problem: During deserialization, the private state of the object is restored, to avoid compromising a class, you must provide either that –

- the sensitive state of an object must not be restored from the stream or
- that it must be reverified by the class.

Solution

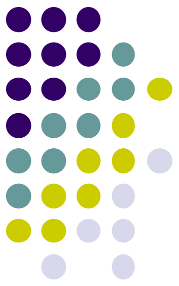
- mark fields that contain sensitive data as *private transient*. transient and static fields are not serialized or deserialized
- Particularly sensitive classes should not be serialized. To accomplish this, the object should not implement either the *Serializable* or *Externalizable* interface.
- Some classes may find it beneficial to allow writing and reading but to specifically handle and revalidate the state as it is deserialized. The class should implement writeObject and readObject methods to save and restore only the appropriate state.

Java I/O - Compression in Java



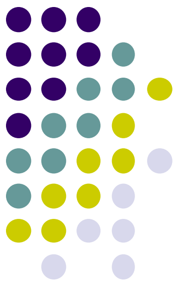
- **java.util.jar**
  - JarInputStream, JarOutputStream
- **java.util.zip**
  - ZIPInputStream, ZIPOuputStream
  - GZIPInputStream, GZIPOutputStream

Java I/O - Example: Creating ZIP file (1)



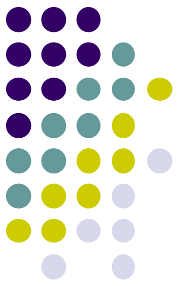
```
String[] filenames = new String[]{"filename1", "filename2"};
byte[] buf = new byte[1024];

try {
  String outFilename = "outfile.zip";
  ZipOutputStream out = new ZipOutputStream(
    new FileOutputStream(outFilename));

  for (int i=0; i<filenames.length; i++) {
    FileInputStream in = new FileInputStream(filenames[i]);

    // <komprese souboru>

    in.close();
  }
  out.close();
} catch (IOException e) {}
```

Java I/O - Example: Creating ZIP file (2)



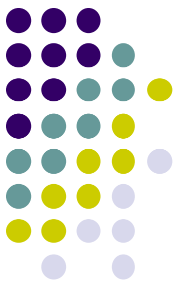
```
// <komprese souboru>

  // Vytvoření nové výstupní položky
  out.putNextEntry(new ZipEntry(filenames[i]));

  // Přenos obsahu souboru
  int len;
  while ((len = in.read(buf)) > 0) {
    out.write(buf, 0, len);
  }

  // Uzavření výstupní položky
  out.closeEntry();
```

Java I/O - Example: Using ZIP file



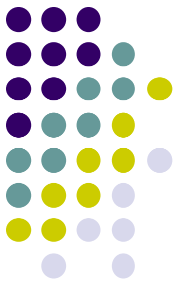
```
try {
    ZipFile zf = new ZipFile(zipFileName);

    for (Enumeration entries = zf.entries();
            entries.hasMoreElements();) {

            String zipEntryName =
            ((ZipEntry)entries.nextElement()).getName();

            System.out.println("Entry : " + zipEntryName );
    }
} catch (IOException e) {
            e.printStackTrace();
}
```

Java I/O - Example: Extracting ZIP file



```
ZipEntry zipEntry = (ZipEntry)entries.nextElement();
String zipEntryName = zipEntry.getName();  int lastDirSep;
if ( (lastDirSep = zipEntryName.lastIndexOf('/')) > 0 ) {
    String dirName = zipEntryName.substring(0, lastDirSep);
      (new File(dirName)).mkdirs();
}
if (!zipEntryName.endsWith("/")) {
    OutputStream out = new FileOutputStream(zipEntryName);
    InputStream in = zf.getInputStream(zipEntry);
    byte[] buf = new byte[1024];  int len;
    while((len = in.read(buf)) > 0) out.write(buf, 0, len);
    out.close(); in.close();
}
```

Threads - Basics

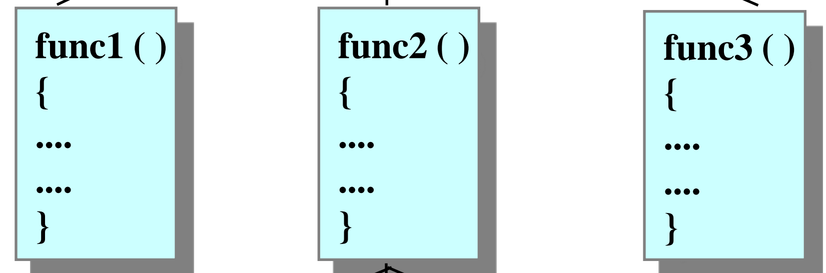


- **Process (task)**
  - Separate "program" with his own memory (address space)
  - Based on operating system
    - Operating system is responsible for process execution.
- **Multitasking -** operation system ability to perform several processes at the same time.
- **Thread**
  - „light waited process"
  - One process may be composed from several threads.
    - Thread's creation is much faster.

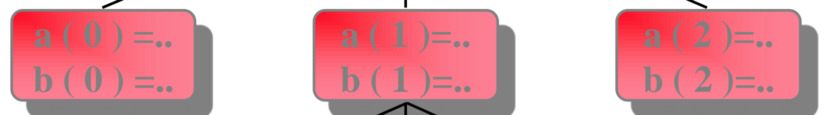
Threads - Level of parallelism

Sockets/ PVM/MPI

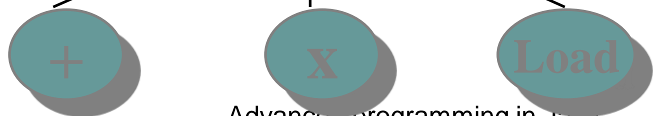
Threads



Compilers



CPU



Code-Granularity

Code Item

Large grain (task level)

Program

Medium grain (control level)

Function (thread)

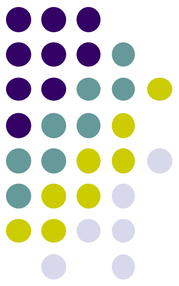
Fine grain (data level)

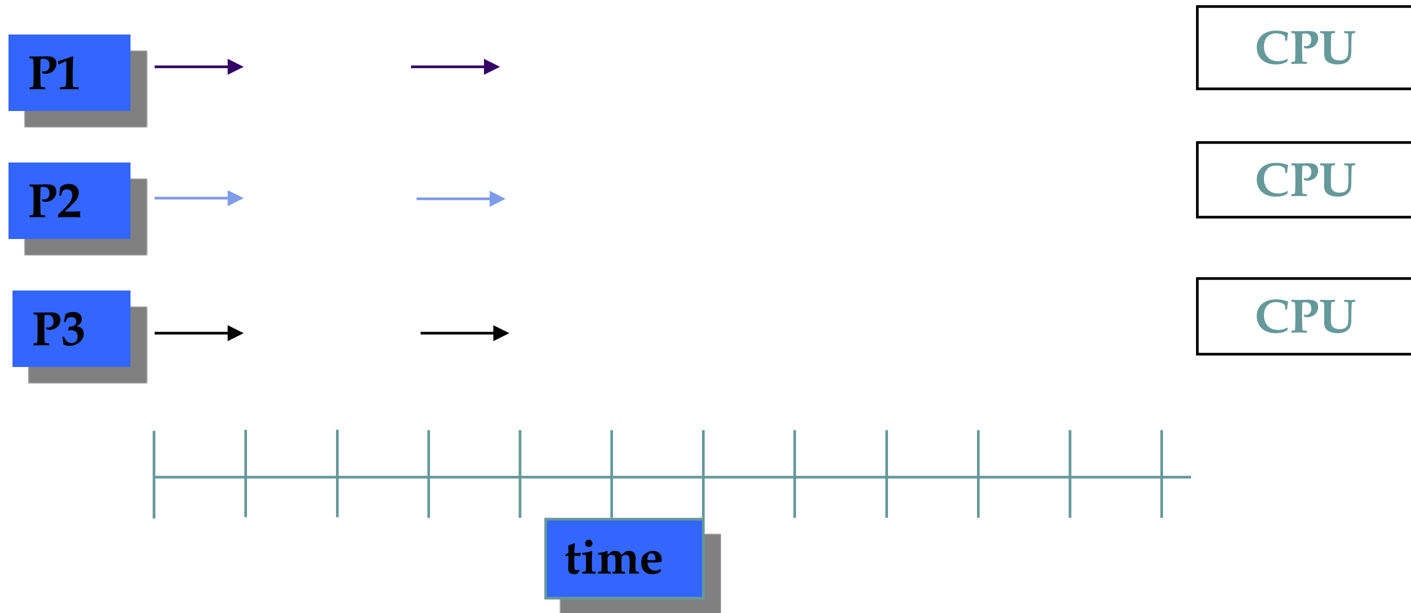
Loop (Compiler)

Very fine grain (multiple issue)

With hardware

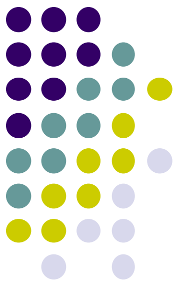
Threads - Execution of multi-thread applications(1)



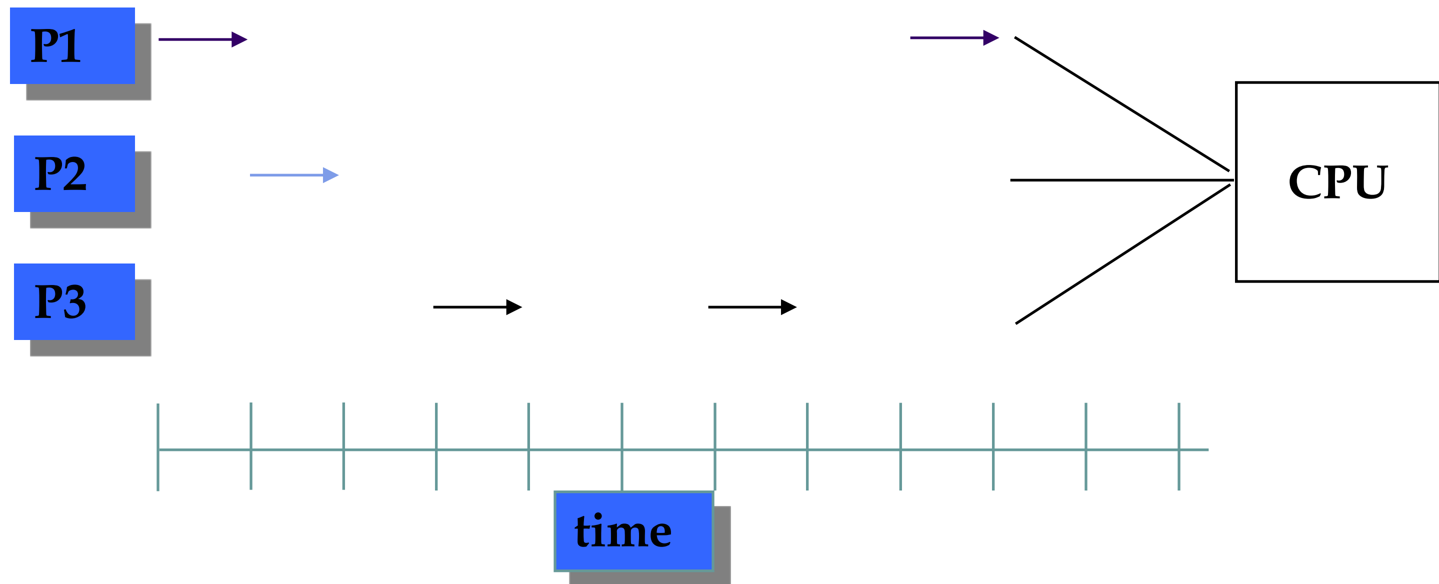


amount of running threads <= amount of CPUs

Threads - Execution of multi-thread applications (2)

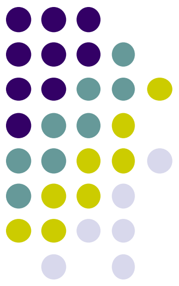


- Concurrent thread execution



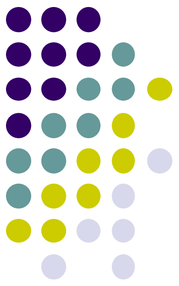
amount of running threads > amount of CPUs

Threads - Creating concurrent applications(1)



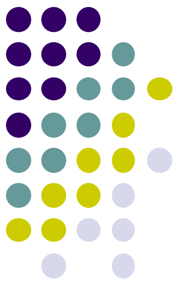
- We can use threads in Java.
- Threads are executed by Java Virtual Machine.
  - They are executed in parallel if possible.

Threads - Creating concurrent applications(2)



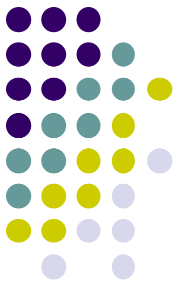
- Thread properties in Java
  - Thread execution starts at specific point of program (*main method*).
  - Instructions are executed one by one with respect to source code.
  - Threads can cooperate together, but they are executed separately.
  - Every thread can access programs data (with respect to Java security rules).
    - **Local properties** – are accessible within a method only.
    - **Instance and static properties** – are shared between threads.

Threads - Thread Creation



- Every class can be a starting point of a new thread. It must:
  - Implement interface `java.lang.Runnable;`
  - Or extends class `java.lang.Thread.`
- Start up point is the `run()` method in both cases.

Threads - Extension of class Thread



- Your class must extend class Thread and re-implement method run().

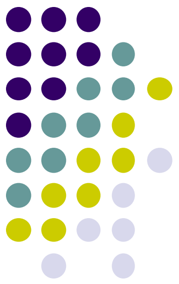
```
class MyThread extends Thread
{
    public void run()
   {
      // thread body
   }
}
```

- Thread’s creation:

```
MyThread thr = new MyThread();
```

- Running created thread:

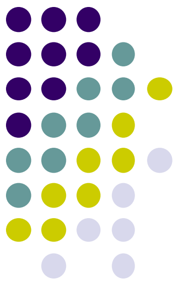
```
thr.start();
```



Threads - Example

```
class MyThread extends Thread {        // thread
        public void run() {
                System.out.println(" this thread is running ... ");
        }
} // end class MyThread

class ThreadEx1 {                      // using thread
        public static void main(String [] args  ) {
           MyThread t = new MyThread();
           //methods start predefined method run
            t.start();
        }
}
```

Threads - Runnable interface



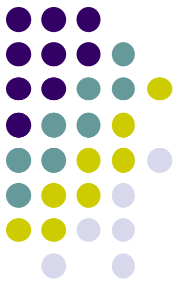
```
class MyThread implements Runnable
{
   .....
   public void run()
   {
      // tělo vlákna
   }
}
```

- Thread's creation:

```
MyThread myObject = new MyThread();
Thread thr1 = new Thread( myObject );
```

- Thread's execution:

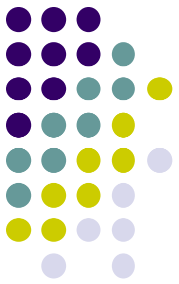
```
thr1.start();
```



Threads - Example

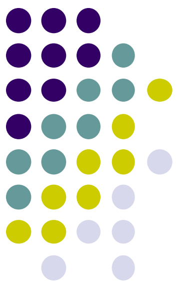
```
class MyThread implements Runnable  {
        public void run() {
                System.out.println(" this thread is running ... ");
        }
}
class ThreadEx2 {
        public static void main(String [] args  ) {

                Thread t = new Thread(new MyThread());

                t.start();
        }
}
```

Threads - Thread class



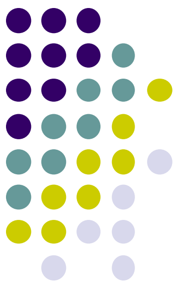
- Basic properties:
  - Constructors
    - public Thread()
    - public Thread(Runnable target)
    - public Thread(String name)
    - public Thread(Runnable target, String name)
  - Basic methods
    - public void start()
    - public void run()

Threads - Starting threads



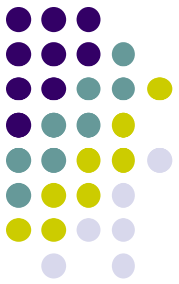
- Invoking its `start` method causes an instance of class Thread to initiate its `run` method
- A Thread terminates when its `run` method completes by either returning normally or throwing an unchecked exception
- Threads are not restartable, even after they terminate
- `isAlive` returns true if a thread has been started by has not terminated

Threads - More Thread methods



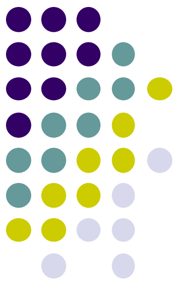
- `Thread.currentThread` returns a reference to the current Thread
- `Thread.sleep(long msecs)` causes the current thread to suspend for at least `msecs` milliseconds
- `Thread.interrupt` is the preferred method for stopping a thread (not `Thread.stop`)

Threads - Priorities



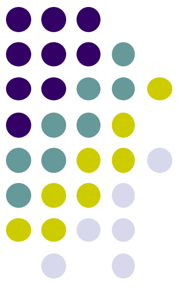
- Each Thread has a priority, between `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY` (from 1 to 10)
- Each new thread has the same priority as the thread that created it
- The initial thread associated with a main by default has priority `Thread.NORM_PRIORITY` (5)
- `getPriority` gets current Thread priority, `setPriority` sets priority
- A scheduler is generally biased to prefer running threads with higher priorities (depends on JVM implementation)

Threads - The “run queue” of runnable threads



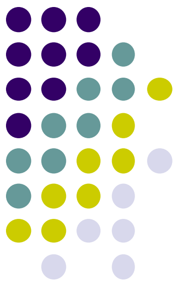
- The Java language specification does not specify how Java is supposed to choose the thread to run if there are several runnable threads of equal priority.
- One possibility – pick a thread and run it until it completes, or until it executes a method that causes it to move into a non-running state.
- Another possibility – “time slicing”: pick a thread and run it for a short period of time. Then, if it is not finished, suspend it and pick another thread to run for the same period of time.

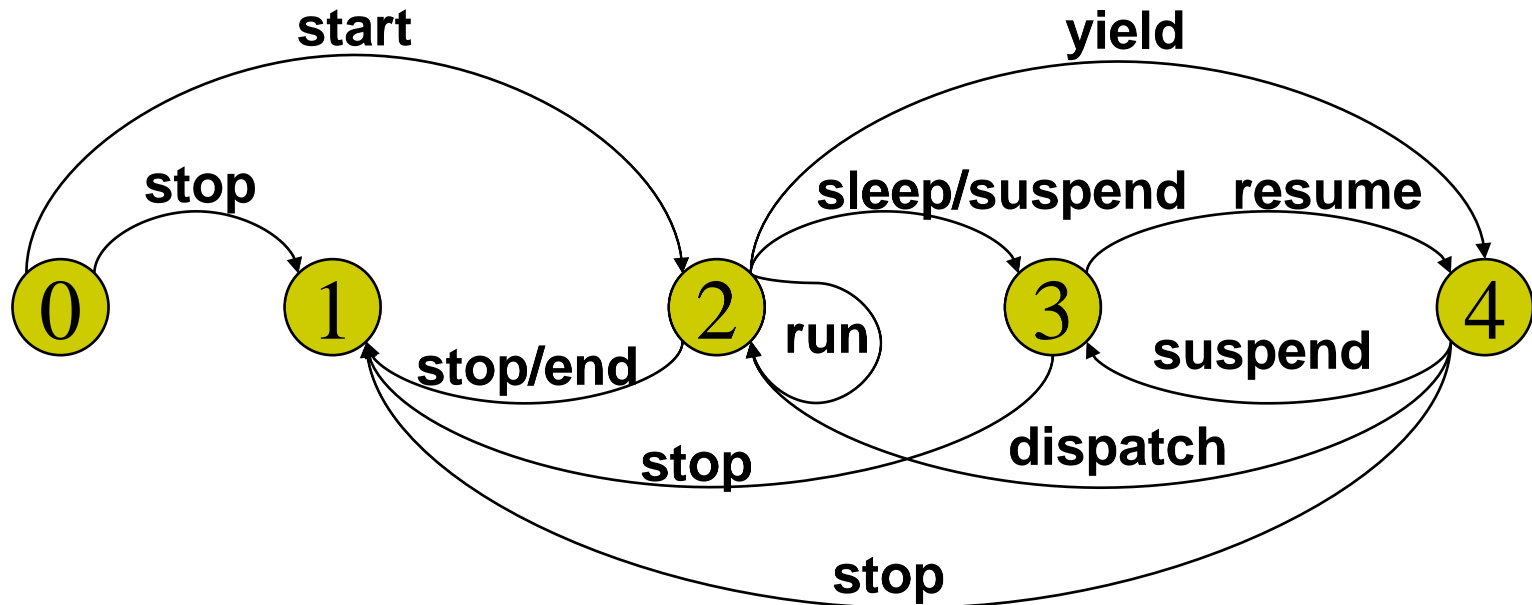
Threads - Thread States and Scheduling



- A Java thread can be in new, runnable, running, suspended, blocked, suspended-blocked and dead.
- The Threads class has methods that move the thread from one state to another.

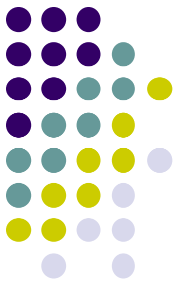
Threads - Thread/process states





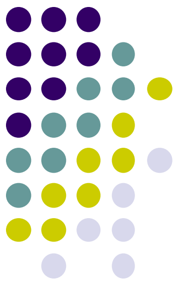
1 – terminated
3 – suspended
2 – running
4 - runnable

Threads - Thread states (1)



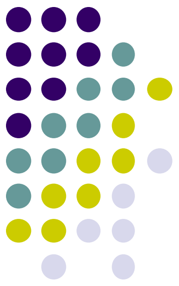
- New state – a Thread newly created.
- Runnable – after being started, the Thread can be run. It is put into the "run queue" of Threads and waits its turn to run. "Runnable" does not mean "running".
- Running – the thread is executing its code. On a uniprocessor machine, at most one thread can run at a time.

Threads - Thread states (2)



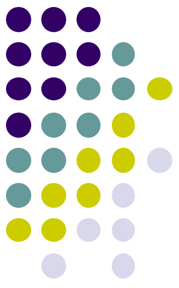
- Blocked – the thread is waiting for something to happen
  It is waiting for an i/o operation it is executing to complete
  It has been told to sleep for a specified period of time through the sleep method
  It has executed the wait() method and will block until another thread executes a notify() or notifyAll() method.
- It will return to runnable state after sleeping, notifying, etc.

Threads - Thread states (3)



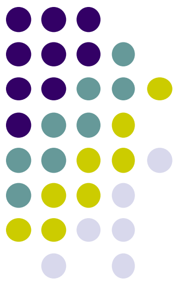
- Dead
  - The final state. After reaching this state the Thread can no longer execute.
  - A thread can reach this state after the run method is finished, or by something executing its stop() method.
  - Threads can kill themselves, or a thread can kill another thread.

Threads - Thread’s life cycle conclusion



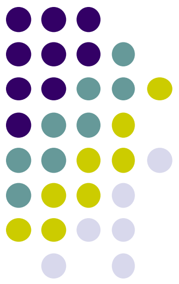
- Basic Thread’s states: *Initial*, *Runnable*, *Not Runnable a Dead*.
- Thread’s methods that affects his life cycle.
  - `public void start()`
  - `public void run()`
  - `public static void sleep(long milisekund)`
  - `public boolean isAlive()`
  - `public void join()`
  - `public void interrupt()`
  - `public boolean isInterrupted()`
  - `public static void yield()`
  - `public Thread.state getState()`

Threads - join( ) Method



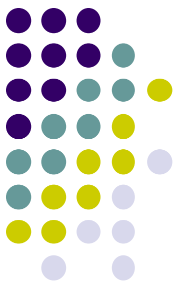
- A call to t1.join( ) causes the current thread to block until Thread t1 terminates
- Throws InterruptedException
- main( ) can join on all threads it spawns to wait for them all to finish
- Optional timeout parameter (milliseconds):
  - `t1.join( 2000 );`

Threads - Daemon Threads



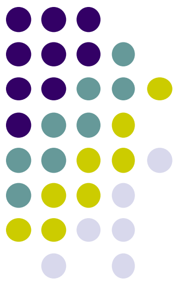
- by themselves do not keep a VM alive
- call setDaemon(true)
  - call must occur before calling start( ); otherwise, an IllegalThreadStateException is thrown
- Thread's default daemon status is the same as the thread that spawned it
- Call isDaemon( ) to see if thread is a daemon

Threads - Concurrency



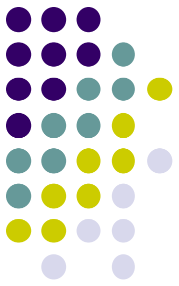
- An object in a program can be changed by more than one thread
  - Q: Is the order of changes that were preformed on the object important? Can it be performed at the same time?
- A race condition – the outcome of a program is affected by the order in which the program's threads are allocated CPU time
  - Two threads are simultaneously modifying a single object
  - Both threads “race” to store their value

Threads - Critical Section



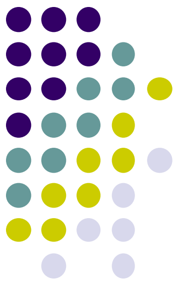
- Section of program that must be executed exclusively by one thread only.
  - Java allows mutual exclusion on objects
    - Acquires the object's *lock*. (Another way of saying “completing the preprotocol”.)
  - `synchronized(obj) { code; }` means that no other `synchronized(obj)` block can be executed simultaneously with `code`.
  - Java use Monitors for locking objects.

Threads - Example



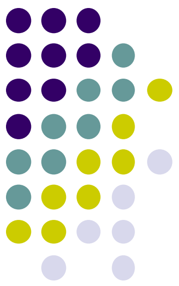
```
public class BankAccount {

    private float balance;

    public synchronized void deposit(float amount) {
        balance += amount;
    }

    public synchronized void withdraw(float amount) {
        balance -= amount;
    }
}
```

Threads – Monitors (1)



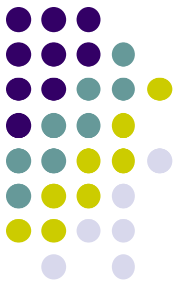
- Each object has a "monitor" that is a token used to determine which application thread has control of a particular object instance
- In execution of a synchronized method (or block), access to the object monitor must be gained before the execution
- Access to the object monitor is queued

Threads - Monitor (2)



- Entering a monitor is also referred to as locking the monitor, or acquiring ownership of the monitor
- If a thread *A* tries to acquire ownership of a monitor and a different thread has already entered the monitor, the current thread (*A*) must wait until the other thread leaves the monitor

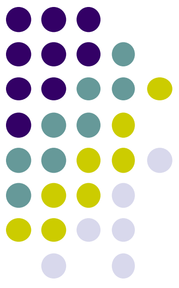
Threads - Java Locks are Reentrant



- Is there a problem with the following code?

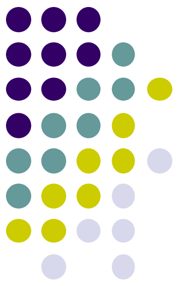
```
public class Test {
        public synchronized void a() {
                b();
                System.out.println("I am at a");
        }
        public synchronized void b() {
                System.out.println("I am at b");
        }
}
```

Threads - The wait() Method (1)



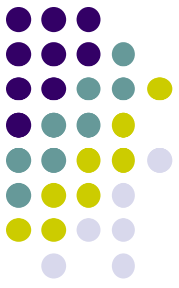
- The **wait()** method is part of the **java.lang.Object** interface
- It requires a lock on the object's monitor to execute
- It must be called from a synchronized method, or from a synchronized segment of code. Why?

Threads - The wait() Method (2)



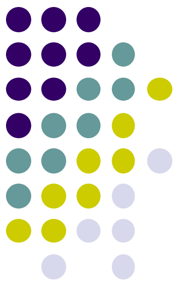
- wait() causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object
- Upon call for wait(), the thread releases ownership of this monitor and waits until another thread notifies the waiting threads of the object

Threads - The wait() Method (3)



- **wait()** is also similar to **yield()**
  - Both take the current thread off the execution stack and force it to be rescheduled
- However, **wait()** is not automatically put back into the scheduler queue
  - **notify()** must be called in order to get a thread back into the scheduler's queue

Threads - Wait and Notify: Code



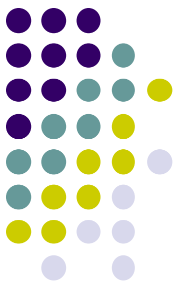
- Consumer:

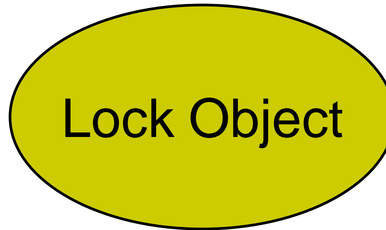
```
synchronized (lock) {
  while (!resourceAvailable()) {
    lock.wait();
  }
  consumeResource();
}
```

- Producer:

```
produceResource();
synchronized (lock) {
  lock.notifyAll();
}
```

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock) {
5.   lock.notify();
6. }
```

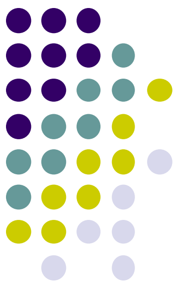
7. Reacquire lock

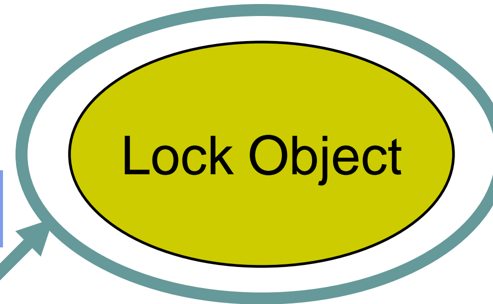
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

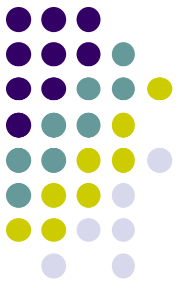
```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence



```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock) {
5.   lock.notify();
6.}
```

Lock Object

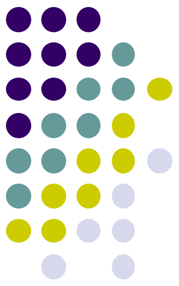
7. Reacquire lock

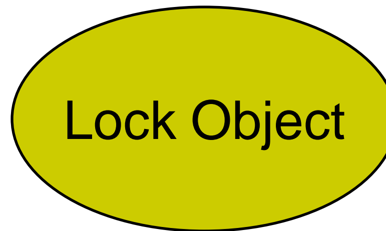
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

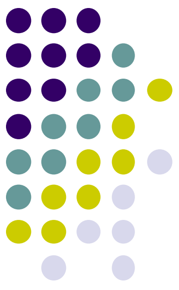
```
3. produceResource()
4. synchronized(lock) {
5.   lock.notify();
6.}
```

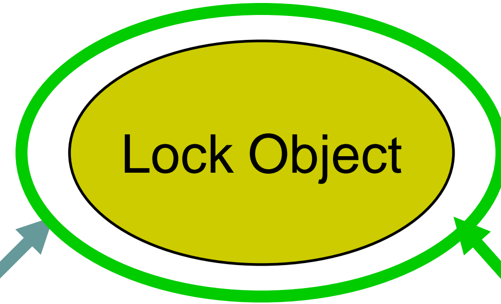
7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6. }
```

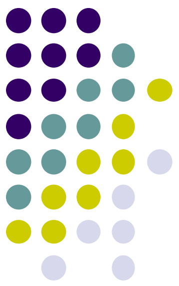
7. Reacquire lock

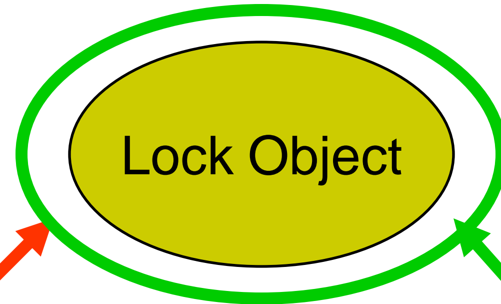
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

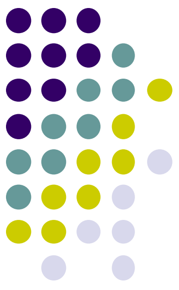
```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6. }
```

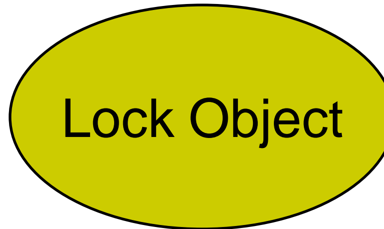
7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6.}
```

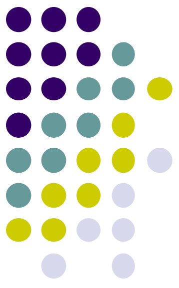
7. Reacquire lock

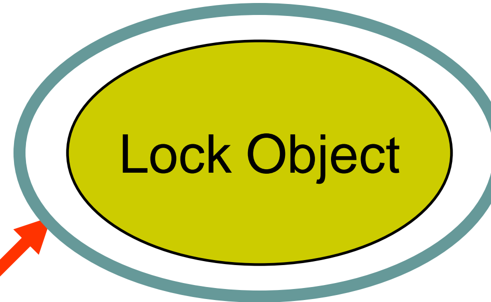
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6.}
```

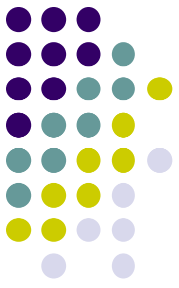
7. Reacquire lock

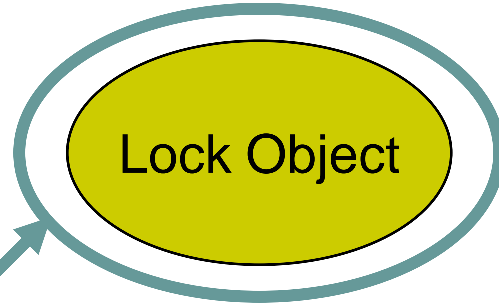
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6.}
```

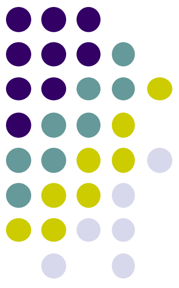
7. Reacquire lock

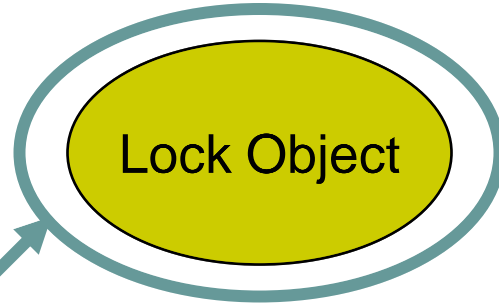
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence






```
1. synchronized(lock){
2.   lock.wait();
9.   consumeResource();
10. }
```

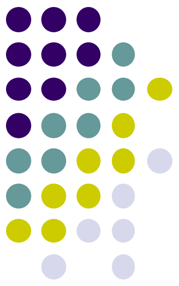
```
3. produceResource()
4. synchronized(lock) {
5.   lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

Producer Thread

Threads - Wait/Notify Sequence



Lock Object

```
1. synchronized(lock){
2.    lock.wait();
9.    consumeResource();
10. }
```

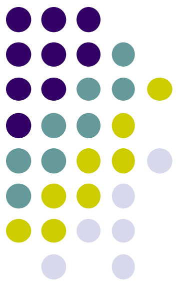
```
3. produceResource()
4. synchronized(lock) {
5.    lock.notify();
6.}
```

7. Reacquire lock
8. Return from wait()

Consumer Thread

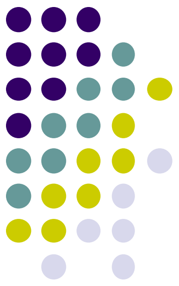
Producer Thread

Threads - Example - Producer



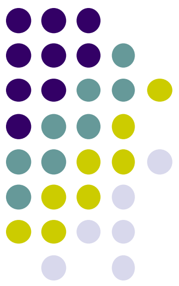
```
class Producer extends Thread{

    private Pool pool;
     public Producer(Pool pool) {
        this.pool=pool;
    }

    public void run() {
        for(int i=0;i<10;i++) {
            System.out.println("Produced item: "+i);
            pool.putItem(i);
            try{
                Thread.sleep(new java.util.Random().nextInt(1000));
            }catch (InterruptedException e) {}
        }
    }
}
```

Threads - Example - Customer



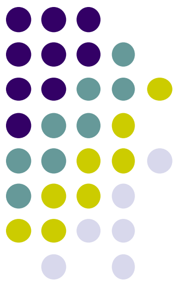
```
class Customer extends Thread{
    private Pool pool;
    private String name;
    public Customer(Pool pool,String name) {
        this.pool=pool;
        this.name=name;
    }
    public void run() {
        for(int i=0;i<5;i++) {
            int tmp=pool.getItem();
            System.out.println(
             name+": Consumed item: "+tmp);
        }
    }
}
```

Threads - Example - Pool



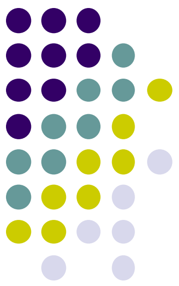
```
class Pool {
    private int item;
    private boolean full = false;

    public synchronized void putItem(int item) {
        while(full) {
            try{
                wait();
            }catch(InterruptedException e){ }
        }
        this.item=item;
        full=true;
        notifyAll();
  }
```

Threads - Example - Pool



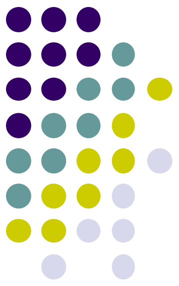
```
    public synchronized int getItem() {
        while(!full) {
            try{
                wait();
            }catch(InterruptedException e) {}
        }
        int tmp= this.item;
        this.full=false;
        notifyAll();
        return tmp;
    }
}
```

Threads - Example - main



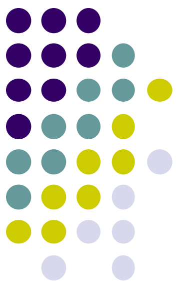
```
public static void main(String[] args) {
        Pool pool = new Pool();
        Producer producer=new Producer(pool);
        Customer consumer1=new Customer(pool,"A");
        Customer consumer2=new Customer(pool,"B");
        consumer1.start();
        consumer2.start();
        producer.start();
}
```

Threads - Example - output



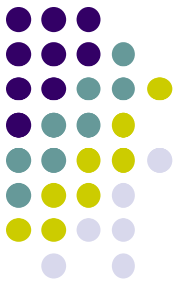
```
Produced item: 0
A: Consumed item: 0
Produced item: 1
B: Consumed item: 1
Produced item: 2
A: Consumed item: 2
Produced item: 3
B: Consumed item: 3
Produced item: 4
A: Consumed item: 4
Produced item: 5
A: Consumed item: 5
Produced item: 6
B: Consumed item: 6
Produced item: 7
A: Consumed item: 7
Produced item: 8
B: Consumed item: 8
Produced item: 9
B: Consumed item: 9
```

Threads - Locks and Pre-Java 5 Approach



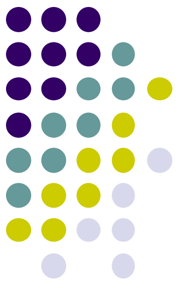
- Each instance of the Java Object class has an object-lock
- Use the *synchronized* keyword for a method
  - Or block of code
  - When entering that method, that thread owns the lock
  - When leaving that method, lock is released
- Condition: something that allows coordination
  - wait() – sleep until the condition for that object becomes true
  - notifyAll() – tell other threads the condition is true

Threads – "New" Java Concurrency Library



- What was just shown is not good design (some argue it's truly broken)
- In Java 5, new approach and library support
  - More like C#, by the way
  - java.util.concurrent
- Lock objects (an interface)
  - Lock has lock() and unlock() methods
- Conditions objects (more than one)
  - Available from a Lock object
  - Condition has signalAll() and await() methods

Threads - Using Java 5 Lock and Conditions



- Define objects in Queue class:

```
private Lock queueLock = new ReentrantLock();
private Condition spaceAvailable =
    queueLock.newCondition();
```

- Need to check a condition?

```
while ( unable to proceed )
    spaceAvailable.await();
// now proceed
```

- Some place else:

```
spaceAvailable.signalAll();
```