new features in J2SE 5.0

- J2SE 5.0 released in September 2004

- new features in language & core libraries
  - generics
  - enums
  - annotations
  - autoboxing, for-each, loop, static import, varargs
  - concurrency utilities
agenda

- generics
- enums
- annotations
- autoboxing, for-each loop, static import, varargs
- concurrency utilities

purpose of generics

- generic types needed for collections
  - implementation is independent of the contained elements
  - use a `generic List` instead of `Int List` and `String List`

- traditional technique for generic types in Java:
  - implementation in terms of `Object` references

- side effects:
  - no collection of primitive types
    - use wrapper types and autoboxing
  - no homogeneous collections
    - lots of casts required
use of non-generic collections

- no homogeneous collections
  - lots of casts required
- no compile-time checks
  - late error detection at runtime

```java
LinkedList list = new LinkedList();
list.add(new Integer(0));
Integer i = (Integer) list.get(0);
String s = (String) list.get(0);
```

casts required

fine at compile-time, but fails at runtime

use of generic collections

- collections are homogeneous
  - no casts necessary
- early compile-time checks
  - based on static type information

```java
LinkedList<Integer> list = new LinkedList<Integer>();
list.add(new Integer(0));
Integer i = list.get(0);
String s = list.get(0);
```

compile-time error
benefits of generic types

- increased expressive power
- improved type safety
- explicit type parameters and implicit type casts

definition of generic types

```java
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

    usage
    - in signatures of methods
    - for declaration of fields and variables
    - as type argument to other generic types
```
requirements to a type parameter (i)

- LinkedList does not require anything of its element type
  - passes around references
  - no object access

```java
class LinkedList<A> implements Collection<A> {
    protected Node head = null, tail = null;
    ...
    public void add (A elt) {
        if (head == null) {
            head = new Node(elt);
            tail = head;
        } else {
            tail.next = new Node(elt);
            tail = tail.next;
        }
    }
}
```

requirements to a type parameter (ii)

- Hashtable needs methods from Object
  - present in all types
  - not a special requirement

```java
public class Hashtable<K,V> { 
    private static class Entry<K,V> { ... }
    private Entry<K,V>[] table;
    ...
    public Data get(K key) {
        int hash = key.hashCode();
        for (Entry<Key,Data> e = table[hash & hashMask];
             e != null; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                return e.value;
            }
        }
        return null;
    }
}
```
requirements to a type parameter (iii)

- **TreeMap** need more than **Object** methods

```java
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) {
        Entry<K, V> p = root;
        K k = key;
        while (p != null) {
            int cmp = k.compareTo(p.key);
            if (cmp == 0) return p;
            else if (cmp < 0) p = p.left;
            else p = p.right;
        }
        return null;
    }
    ...
}
```

- **Bounds =** classes or interfaces that a type variable extends and implements
- **Purpose:**
  - make available no-static methods of a type variable
  - gives no access to constructors or static methods
- **Syntax:**
  
  ```java
  TypeVariable extends Superclass &
  Interface1 & Interface2 & ... & Interface_n
  ```

```java
class Pair<A extends Comparable<A> & Cloneable, B extends Comparable<B> & Cloneable>
    implements Comparable<Pair<A, B>>, Cloneable {
    ...
}
using generic types

• 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

concrete instantiation

• type argument is a concrete type

```java
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

```java
List<File> targetDir = new LinkedList<File>();
... fill list with File objects ...
printDirectoryNames(targetDir);
```
raw type

- no type argument specified

```java
void printDirectoryNames(Collection files) {
    for (Iterator it = files.iterator(); it.hasNext(); ) {
        File f = (File) it.next();
        if (f.isDirectory())
            System.out.println(f);
    }
}
```

- permitted for compatibility reasons
  - permits mix of non-generic (legacy) code with generic code

```java
List<File> targetDir = new LinkedList<File>();
... fill list with File objects ...
prientDirectoryNames(targetDir);
```

wildcard instantiation

- type argument is a wildcard

```java
void printElements(Collection<? c) {
    for (Object e : c)
        System.out.println(e);
}
```

- a wildcard stands for a family of types
  - bounded and unbounded wildcards supported

```java
Collection<File> targetDir = new LinkedList<File>();
... fill list with File objects ...
prientElements(targetDir);
```
wildcards

- 3 families of types:
  
  - unbounded wildcard \(?\)
    - all types
  
  - upper-bound wildcard \(? \text{extends} \text{Supertype}\)
    - all types that are subtypes of \text{Supertype}
  
  - lower-bound wildcard \(? \text{super} \text{Subtype}\)
    - all types that are supertypes of \text{Subtype}

definition of generic methods

```
class Collections {
    public static <A extends Comparable<A>> A max (Collection<A> xs) {
        Iterator<A> xi = xs.iterator();
        A w = xi.next();
        while (xi.hasNext()) {
            A x = xi.next();
            if (w.compareTo(x) < 0) w = x;
        }
        return w;
    }
}
```

- constructors may be generic, too
use of generic methods

• no special invocation syntax
  – type arguments are inferred from actual arguments

```java
final class Test {
    public static void main (String[ ] args) {
        LinkedList<Byte> byteList = new LinkedList<Byte>();
        byteList.add(new Byte((byte)0));
        byteList.add(new Byte((byte)1));

        Byte y = Collections.max(byteList);
    }
}
```

wrap-up

• generics increase type safety and expressiveness
• generic types & generic methods
• type variables = placeholder for unknown type
• bounds = give access to non-static methods
• instantiation with concrete type and wildcard
• raw type permitted for compatibility
• implicit type inference for generic methods
agenda

• generics
• enums
• annotations
• autoboxing, for-each loop, static import, varargs
• concurrency utilities

enum type

• enum types in Java 5.0:
  
  ```java
  public enum Season { winter, spring, summer, fall }
  ```

• type-safe alternative to old-style static finals
  
  ```java
  public class Season {
    static final int WINTER; static final int SPRING; ... }
  ```

• design goals:
  – compile-time type safety
  – performance comparable to int constants
  – typesafe constants aren't compiled into clients
    • you can add, reorder or remove constants without recompiling clients
  – printed values are informative
  – enum constants can be used in collections, e.g. as HashMap keys
### Enums with Fields - Use in Switch

```java
public enum Coin {
    penny(1), nickel(5), dime(10), quarter(25);
    private final int value;
    private Coin(int value) { this.value = value; }
    public int value() { return value; }
}
```

```java
private enum CoinColor { copper, nickel, silver }
```

```java
CoinColor color(Coin c) {
    if (c == null) throw new NullPointerException();
    switch (c) {
    case Coin.penny: return CoinColor.copper;
    case Coin.nickel: return CoinColor.nickel;
    case Coin.dime: return CoinColor.silver;
    case Coin.quarter: return CoinColor.silver;
    }
    throw new AssertionError("Unknown coin: " + c);
}
```

### Methods per Enum Value

```java
public abstract enum Operation {
    plus {double eval(double x, double y) { return x + y; }},
    minus {double eval(double x, double y) { return x - y; }},
    times {double eval(double x, double y) { return x * y; }},
    div {double eval(double x, double y) { return x / y; }};

    // perform arithmetic operation represented by this constant
    abstract double eval(double x, double y);
}
```

```java
void f(double x, double y) {
    for (Operation op : Operation.values()) {
        System.out.println(x + " +" + op + " +y = " + op.eval(x, y));
    }
}
```
agenda

- generics
- enums
- annotations
- autoboxing, for-each loop, static import, varargs
- concurrency utilities

program annotation facility

- allows developers
  - to define custom *annotation types*
  - to *annotate* fields, methods, classes, etc. with *annotations* corresponding to these types

- allow tools to read and process the annotations
  - no direct effect on semantics of a program
  - e.g. tool can produce additional Java source files or XML documents related to the annotated program
sample usage

- annotated class

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

- corresponding definition of annotation type

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

- reading an annotation via reflection

```java
String copyrightHolder = OscillationOverthruster.class.
getAnnotation(Copyright.class).value();
```

retention

- it makes little sense to retain all annotations at run time
  - would increase run-time memory-footprint

- annotations can have different lifetime:

  SOURCE:
  - discarded after compilation

  CLASS:
  - recorded in the class file as signature attributes
  - not retained until run time

  RUNTIME:
  - recorded in the class file and retained by the VM at run time
  - may be read reflectively
agenda

- generics
- enums
- annotations
- autoboxing, for-each loop, static import, varargs
- concurrency utilities

autoboxing

- automatic conversion of data of primitive type to corresponding wrapper type and vice versa

```java
int[] values = { 5, 13, 12, 40, 30 };  
Collection<Integer> collection = new ArrayList<Integer>();  
for (int val : values)  
{ collection.add(val); }
```

```java
Collection<Integer> collection;  
int summe = 0;  
for (Integer val : collection)  
{ summe += val; }
```
enhanced for-loop

• standard idiom to iterate over a collection is somewhat verbose:

```java
for (Iterator i = c.iterator(); i.hasNext(); ) {
    String s = (String) i.next();
    ...  
}
```

• new form of for loop specifically designed for iteration over collections and arrays:

```java
for (String s : c) {
    ...  
}
```

static import

• allow importation of static methods and fields
  – in the manner that classes and interfaces can now be imported
• example:
  – methods from `java.lang.Math` are static methods
  – must be named `Math.abs(x)`, `Math.sqrt(x)`, `Math.max(a, b)`

```java
import static java.lang.Math.*;
...

x = sqrt(5);
y = abs(y);
z = max(x, y);
```
varargs

- varargs allow specification of a sequence of arguments
- have convenient invocation syntax

```java
public class Formatter {
    public void format(String format, Object... args) {
    }
    formatter.format("%d bytes in %d seconds (%.2f KB/s)\n", 512, 60, ((double)(512 / 1024) / (double)60));
}
```

agenda

- generics
- enums
- annotations
- autoboxing, for-each loop, static import, varargs
- concurrency utilities
situation before JDK 5.0

- concurrency API more or less stable since JDK 1.0
  - but big changes with JDK 5.0 – why? what was missing?

- high-level concurrency support
  - very few active objects in the JDK
    - most desired: thread pool
      - you find countless examples in Java books, magazines on the web, etc.
  - no high-level synchronization / communication abstractions between threads
    - e.g. a blocking queue
    - you always had to start with `wait() / notify()`

Callable vs. Runnable

- similar to `Runnable`, but
  - returns a result
  - throws an exception if it fails
  - still no parameter

- defined as:

  ```java
  public interface Callable<V> {
      V call() throws Exception;
  }
  ```
future pattern

• Future = abstraction that lets you obtain the result of an asynchronous activity
  – called future or IOU (for "I owe you a result")

• FutureTask = façade that combines
  – an asynchronous task (Runnable/Callable)
  – waiting and retrieval of results from the task
  – interruption of task and check for cancellation/success

```java
class MyCallableRunner {
    public MyCallableRunner(final int cnt) {
        Callable<Long> c = new Callable<Long>() {
            public Long call() {
                long start = new Date().getTime();
                for (int i = 0; i < cnt; i++) {
                    System.out.println("-");
                }
                return new Long(new Date().getTime() - start);
            }
        };
        FutureTask<Long> f = new FutureTask<Long>(c);
        Thread t = new Thread(f);
        t.setDaemon(false);
        t.start();
        System.out.println("It took " + f.get() + " msec to print " + cnt + " times the character: \"-\".");
    }
}
```
queue - description

- data structure based on first-in-first-out (fifo) policy
  - oppose to a stack: first-in-last-out (filo)
- blocking queue
  - most commonly used exchange medium between cooperating threads working according to the producer-consumer-pattern

```java
class Producer implements Runnable {
    private final BlockingQueue queue;
    Producer(BlockingQueue q) { queue = q; }
    public void run() {
        try { while(true) { queue.put(produce()); } }
        catch (InterruptedException ex) { ... handle ... }
        Object produce() { ... }
    }
class Consumer implements Runnable {
    ...
}
class Setup {
    void main() {
        BlockingQueue q = new LinkedBlockingQueue();
        Producer p = new Producer(q);
        Consumer c1 = new Consumer(q);
        Consumer c2 = new Consumer(q);
        new Thread(p).start();
        new Thread(c1).start();
        new Thread(c2).start();
    }
}
```

blocking queue - example
synchronizers

- exchanger
  - synchronization point at which two threads can exchange objects
- barrier
  - allows a set of threads to all wait for each other to reach a common barrier point
- semaphore
  - restricts the number of threads than can access some (physical or logical) resource
- latch
  - allows one or more threads to wait until a set of operations being performed in other threads completes

countdown latch - example

- a pair of classes in which a group of worker threads use two countdown latches:
  - 1st latch: start signal that prevents any worker from proceeding until the driver is ready for them to proceed
  - 2nd latch: completion signal that allows the driver to wait until all workers have completed
• threads often maintained in a pool
  – for performance improvement and/or better control the behavior of the overall system
• class ThreadPoolExecutor implements a thread pool
  – execute a Runnable or submit a Callable task
    - task put into an internal queue
      • queue can be configured: any queue class that implements BlockingQueue
      • special case: size == 0
        – when task is at the beginning of the queue and a pool thread becomes idle
          task is taken from the queue and applied to this pool thread

supplying a task to a pool thread

```java
void execute(Runnable cmd)
```

queue

pool threads
**configuration of ThreadPool Executor**

- important issue: configuration / tuning
  - core pool size
  - maximum pool size
  - internal queue
  - timeout value for pool threads
  - thread factory
  - rejected execution handler
- predefined factory methods
- subclasses can override protected methods

**scatter-gather processing**

- thread pool
  - supports *scatter* phase in a scatter-gather architecture
  - takes *Runnable*s that do not produce a result

- completion service
  - supports *gather* phase in a scatter-gather architecture
  - takes *Callable*s that do produce a result
  - converts them to *FutureTasks*
  - places the *Futures* into a queue
  - from which they can be retrieved for further processing
J2SE 5.0 - new features

architecture of a completion service

- Future submit(Callable cmd)

  task queue

  pool threads

  result queue

usage - example

- run a set of solvers for a certain problem concurrently
  - each solver returns a result
- process the results in a method use()
  - if solver returned a non-null value

```java
void solve(Executor e, Collection<Callable<Result>> solvers)
    throws InterruptedException, ExecutionException {
    CompletionService<Result> ecs = new ExecutorCompletionService<Result>(e);
    for (Callable<Result> s : solvers) ecs.submit(s);
    int n = solvers.size();
    for (int i = 0; i < n; ++i) {
        Result r = ecs.take().get();
        if (r != null) use(r);
    }
```
not mentioned yet ...

- explicit locks
- explicit conditions
- scheduled tasks
- lock-free programming
- memory model issues

wrap-up

- key change in the language: generics
- minor language additions: enums, autoboxing, etc.
- annotations for tool builders
- improved support for concurrent programming