Java Type Conversion

Assume class `B` is a subclass of class `A` (`class B extends A`). In the following piece of code:

```
A a; B b;
a = new A(); b = new B(); a = b;    // line 1
a = new A(); b = new B(); b = a;    // line 2
a = new A(); b = new B(); a = (A)b; // line 3
a = new A(); b = new B(); b = (B)a; // line 4
a = new B(); b = new B(); a = b;    // line 5
a = new B(); b = new B(); b = a;    // line 6
a = new B(); b = new B(); a = (A)b; // line 7
a = new B(); b = new B(); b = (B)a; // line 8
```

which lines would result in an error

1. at compile-time?
2. at run-time?

Assume class `B` is a subclass of class `A` (`class B extends A`). In the following piece of code:

```
A[] a; B[] b;
a = new A[1]; b = new B[1]; a = b;       // line 1
a = new A[1]; b = new B[1]; b = a;       // line 2
a = new A[1]; b = new B[1]; a = (A[])b;  // line 3
a = new A[1]; b = new B[1]; b = (B[])a;  // line 4
a = new B[1]; b = new B[1]; a = b;       // line 5
a = new B[1]; b = new B[1]; b = a;       // line 6
a = new B[1]; b = new B[1]; a = (A[])b;  // line 7
a = new B[1]; b = new B[1]; b = (B[])a;  // line 8
```

which lines would result in an error

1. at compile-time?
2. at run-time?

ArrayStoreException

Assume class `B` and `C` are two subclasses of class `A` (`class B extends A`), (`class C extends A`). What is the problem with the following code:

```
void Foo(A[] a){
        a[0] = new C();
        ...
}

 B[] b = new B[1];
 Foo(b);
```

Assume class `B` is a subclass of class `A` (`class B extends A`). In the following piece of code:

```
List<A> a = new LinkedList<A>();
List<B> b = new LinkedList<B>();
a = b;                          // line 1
a = (List<A>)b;                 // line 2
b = a;                          // line 3
b = (List<B>)a;                 // line 4
```

which lines would result in an error

1. at compile-time?
2. at run-time?

Note:

- generics allow for **parameterized types**
- `List<Integer>` is **NOT** a subtype of `List<Number>`
  (`Integer[]` *is* a subtype of `Number[]` but `ArrayStoreException`)
- `<? extends T>` denotes a subtype of `T`
- `<S extends T>` denotes a subtype `S` of `T`
- `<? super T>` denotes a supertype of `T`
- `List<Integer>` **is** a subtype of `List<? extends Number>`
- `List<Number>` **is** a subtype of `List<? super Integer>`

```
interface MyInterface {}
class Parent {}
class Child extends Parent implements MyInterface {}
class InstanceofExample {
   public static void main(String[] args) {
       Parent obj1 = new Parent();
       Parent obj2 = new Child();

       System.out.println("obj1 instanceof Parent: "
           + (obj1 instanceof Parent));
       System.out.println("obj1 instanceof Child: "
           + (obj1 instanceof Child));
       System.out.println("obj1 instanceof MyInterface: "
           + (obj1 instanceof MyInterface));
       System.out.println("obj2 instanceof Parent: "
           + (obj2 instanceof Parent));
       System.out.println("obj2 instanceof Child: "
           + (obj2 instanceof Child));
       System.out.println("obj2 instanceof MyInterface: "
           + (obj2 instanceof MyInterface));
   }
}
```

What is the output of this program?

Write code that behaves like:

```
return a instanceof b;
```

but does not use the keyword `instanceof`. The code only needs to be functionally correct, not efficient.

Write code that behaves like:

```
return a instanceof b;
```

but does not use the keyword `instanceof`. The code only needs to be functionally correct, not efficient.

```
try {
      return (b)a != null;
}
catch (ClassCastException e) {
      return false;
}
```

or

```
return a != null && b.class.isAssignableFrom(a.getClass());
```

Bounded Generic Wildcard

```
public class A { }   public class B extends A { }   public class C extends A { }
```

```
public void processElements(List<A> elements){   // ?
   for(A a : elements)
       System.out.println(a.getValue());
}
List<A> listA = new ArrayList<A>();
processElements(listA);
List<B> listB = new ArrayList<B>();
processElements(listB);
List<C> listC = new ArrayList<C>();
processElements(listC);
```

```
public static void insertElements(List<A> list){    // ?
    list.add(new A());
    list.add(new B());
    list.add(new C());
}
List<A> listA = new ArrayList<A>();
insertElements(listA);
List<Object> listObject = new ArrayList<Object>();
insertElements(listObject);
```

Any errors in the following code?

```
public class MyLinkedList<T> {
  public static void someMethod(Object o) {
      if (o instanceof T) {   // ?
          ...
      }
      T[]  arr = new T[5];  // ?
      T  item = new T();     // ?
      T obj = new Object();  // ?
      T obj = (T) new Object();  // ?
  }
  ...
}
```

toArray revisit

```
public E[] toArray () {
  E[] a = new E[size];
  Cell<E> c = head;
  for (int i=0; i<size; i++) {
    a[i] = c.element;
    c = c.next;
  }
  return a;
}
```

```
public E[] toArray () {
  E[] a = (E[])new Object[size];
  Cell<E> c = head;
  for (int i=0; i<size; i++) {
    a[i] = c.element;
    c = c.next;
  }
  return a;
}
```

```
public Object[] toArray () {
  Object[] a = new Object[size];
  Cell<E> c = head;
  for (int i=0; i<size; i++) {
    a[i] = c.element;
    c = c.next;
  }
  return a;
}
```

```
MyLinkedList<String> l = new MyLinkedList<>(...);
String[] a = l.toArray();
```

```
@SuppressWarnings("unchecked")
 public T[] toArray(T[] arr){
     if (arr.length < size) {
         // If array is too small, allocate a new one
         arr = (T[])Array.newInstance(arr.getClass().getComponentType(), size);
     }
    Cell<T> cur = head;
    for (int i=0; i<size; i++) {
        arr[i] = cur.element;
        cur = cur.next;
    }

    if (arr.length > size) {
        // If array is too large, set the first unassigned element to null
        arr[size] = null;
     }

    return arr;
}
```

- a method `E[] toArray (E[] a)` is written using **reflection**
  (it uses the existing array or reallocates a larger one)

```
public class TwoThreads {
 static Thread t1, t2;
 public static void main(String[] args) {
   t1 = new Thread() {
      public void run() {
         System.out.println("A");
         try {
             t2.sleep(1000);
         } catch (Exception e) {
             System.out.println("B");
         }
         System.out.println("C");
      }
   };
   t2 = new Thread() {
      public void run() {
         System.out.println("D");
         try {
              t1.wait();
         } catch (Exception e) {
              System.out.println("E");
         }
         System.out.println("F");
      }
   };
   t1.start();  t2.start();
 }
```

```
class PingPong implements Runnable {
  public void run () {
    ...
  }
  ...
}
```

- When `PingPong.run` is called, two threads, *Ping* and *Pong* are created and write "Ping" and "Pong" in turns on the terminal. The threads cooperate through **proper synchronization** (no *busy-wait*)
- Same question with **3 threads**, *Ping*, *Pong* and *Pang*

```
class PingPong0 {
        public static void main(String [] args){
                new P("Ping").start();
                new P("Pong").start();
        }
}
class P extends Thread {
        P (String name) { super(name); }
        public void run () {
                String name = getName();
                synchronized (System.out) {
                        while (true) {
                                // System.out.notify();   will work if placed here
                                try {
                                        System.out.wait();
                                } catch (InterruptedException e) { return; }
                                System.out.notify();
                                System.out.println(name);
                        }
                }
        }
}
```

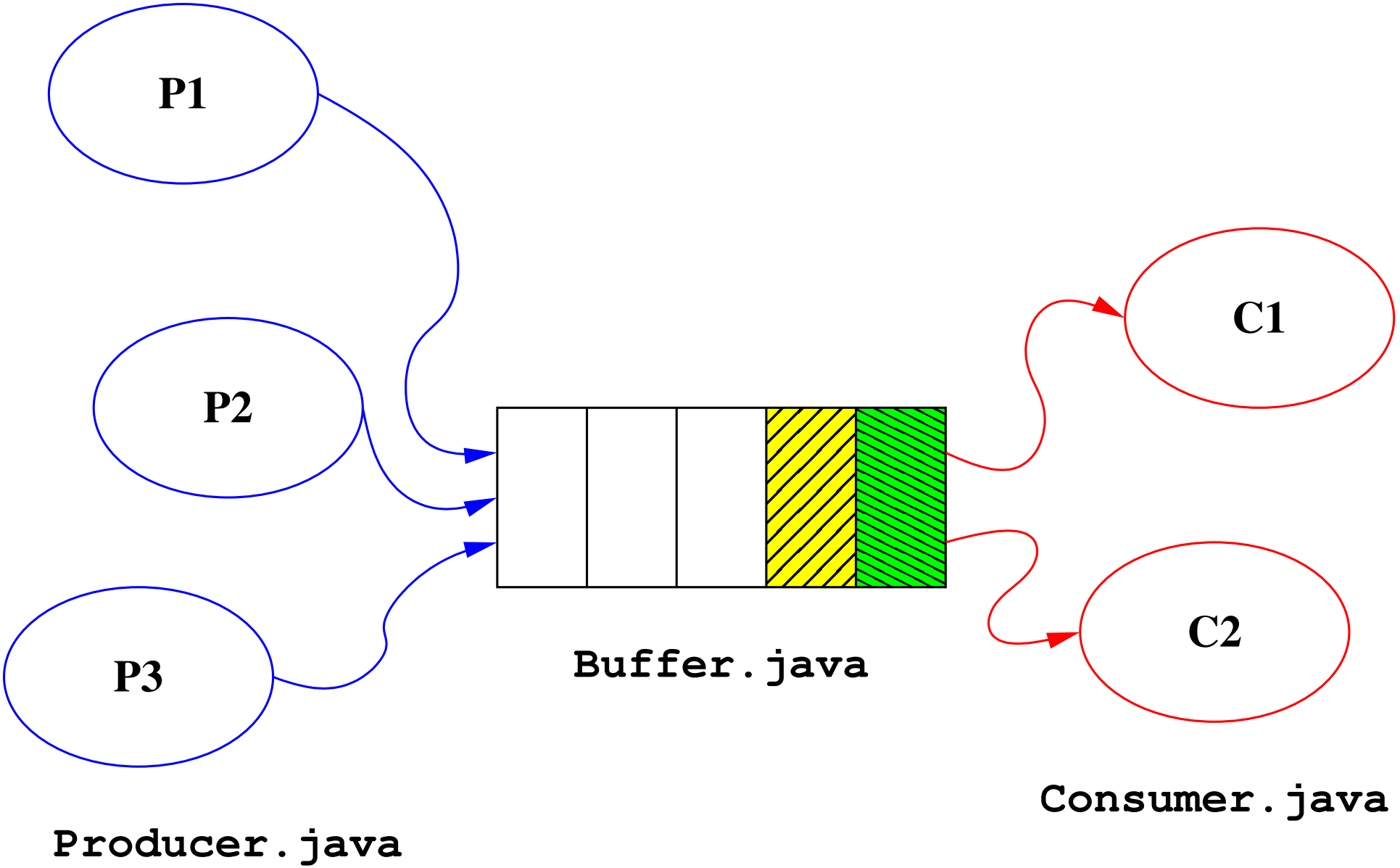
```
class PingPong1 implements Runnable {
   private Turn turn = new Turn();
      public void run () {
        new P("Ping", turn, true).start();  new P("Pong", turn, false).start();
      }
      public static void main(String [] args){ new PingPong1().run(); }
}
class Turn{
      Turn() { turn = true; }
      boolean turn;
}
class P extends Thread {
      P (String name, Turn t, boolean myturn) { super(name); this.t = t; this.myTurn = myturn;
      Turn t;
      boolean myTurn;
      public void run () {
          String name = getName();
          synchronized (System.out) {
                while (true) {
                      if (t.turn != myTurn) {   //  what if spurious wakeup
                          try {
                               System.out.wait();
                          } catch (InterruptedException e) { return; }
                      }
                     System.out.println(name);
                     t.turn = !t.turn;
                     System.out.notify();
     }}}}
```

```
class PingPong implements Runnable {
  private boolean turn;
  private final Object lock = new Object();

  public void run () {
    class P extends Thread {
      P (String name, boolean myTurn) { super(name); this.myTurn = myTurn; }
      boolean myTurn;
      public void run () {
        String name = getName();
        synchronized (lock) {
          while (true) {
            while (turn != myTurn) {
              try {
                lock.wait();
              } catch (InterruptedException e) { return; }
            }
            System.out.println(name);
            turn = !turn;
            lock.notify();
          }
        }
      }
    }
    new P("Ping", true).start(); new P("Pong", false).start();
  }
}
```

```
class PingPongPang implements Runnable {
  private int turn;
  private final Object lock = new Object();

  public void run () {
    class P extends Thread {
      P (String name, int myTurn) { super(name); this.myTurn = myTurn; }
      int myTurn;
      public void run () {
        String name = getName();
        synchronized (lock) {
          while (true) {
            while (turn != myTurn) {
              try {
                lock.wait();
              } catch (InterruptedException e) { return; }
            }
            System.out.println(name);
            turn = (myTurn == 2)? 0 : myTurn + 1;
            lock.notifyAll();
          }
        }
      }
    }
    new P("Ping", 0).start(); new P("Pong", 1).start(); new P("Pang", 2).start();
  }
}
```

```
private final Object[] locks = new Object[] { new Object(), new Object(), new Object() };
...
  class P extends Thread {
  ...
    public void run () {
      String name = getName();
      Object myLock = locks[myTurn];
      synchronized (myLock) {
        while (true) {
          while (turn != myTurn) {
            try {
              myLock.wait();
            } catch (InterruptedException e) {
              return;
            }
          }
          System.out.println(name);
          turn = (myTurn == 2)? 0 : myTurn + 1;
          Object nextLock = locks[turn];
          synchronized (nextLock) { // why?
             nextLock.notify();
          }
        }
      }
    }
```

Case Study: Bounded Buffer
(producer-consumer)



Bounded Buffer Example

```
public class Buffer<E> {

  public final int capacity;
  private final E[] store;
  private int head, tail, size;

  @SuppressWarnings("unchecked")
  public Buffer (int capacity) {
    this.capacity = capacity;
    this.store = (E[])new Object[capacity];
  }
  private int next (int x) {
    return (x + 1) % store.length;
  }
  public synchronized void put (E e) throws InterruptedException {
    while (isFull())
      wait();
    notifyAll();
    store[tail] = e;
    tail = next(tail);
    size++;
  }
```

```
  public synchronized E get () throws InterruptedException {
    while (isEmpty())
      wait();
    notifyAll();
    E e = store[head];
    store[head] = null; // for GC
    head = next(head);
    size--;
    return e;
  }
  public synchronized boolean isFull () {
    return size == capacity;
  }
  public synchronized boolean isEmpty () {
    return size == 0;
  }
}
```

Complete class `Sender` by implementing all the **parallel methods**

```
class Sender {
  ...
  protected void sendFile (File f) { ... }

  // Purely sequential
  public void sendAllFiles (File[] files) {
    for (File f : files) {
      sendFile(f);
    }
  }

  // Purely parallel, asynchronous
  public void sendAllFilesPar (File[] files) { ... }

  // Purely parallel, synchronous
  public void sendAllFilesParWait (File[] files) { ... }

  // Bounded parallelism, new threads
  public void sendAllFilesParWait (final File[] files, int k) { ... }

  // Bounded parallelism, reusing threads
  public void sendAllFilesPoolWait (File[] files, int k) { ... }
}
```

```
// Purely parallel, asynchronous
public void sendAllFilesPar (File[] files) {
  for (final File f : files)
    new Thread("Sender of "+f) {
      public void run () {
        sendFile(f);
      }
    }.start();
}
```

```
// Purely parallel, synchronous
public void sendAllFilesParWait (File[] files) {
  final int n = files.length;
  Thread[] senders = new Thread[n];
  for (int i=0; i<n; i++) {
    final File f = files[i];
    Thread t = senders[i] = new Thread("Sender of "+f) {
        public void run () {
          sendFile(f);
      }
      };
    t.start();
  }
  for (Thread t : senders)
    try {
      t.join();
    } catch (InterruptedException e) {
      return;
    }
}
```

```
// Bounded parallelism, new threads
public void sendAllFilesParWait (final File[] files, int k) {
  Thread[] senders = new Thread[k];
  class Dispatcher {
    int next;
    synchronized File nextFile () {
      if (next == files.length)
        return null;
      return files[next++];
    }
  }
  final Dispatcher d = new Dispatcher();
  for (int i=0; i<k; i++) {
    Thread t = senders[i] = new Thread("Sender "+i) {
        public void run () {
          File f;
          while ((f = d.nextFile()) != null)
            sendFile(f);
        }
      };
    t.start();
  }
  for (Thread t : senders)
    try { t.join(); } catch (InterruptedException e) { return; }
}
```

```
// Bounded parallelism, reusing threads
private final Worker[] workers;
private int activeThreads;

public Sender (int k) { // constructor takes the bound
  workers = new Worker[k];
  for (int i=0; i<k; i++) {
    Thread t = workers[i] = new Worker("Worker "+i);
    t.start();
  }
}
public void terminate () { // to properly terminate workers
  for (Worker w : workers)
    w.interrupt();
}
private synchronized int activeCount () {
  return activeThreads;
}
private synchronized void setActive () {
  activeThreads++;
}
private synchronized void setInactive () {
  activeThreads--;
  if (activeThreads == 0)
    notify();
}
```

```
private class Worker extends Thread {
  public Worker (String name) { super(name); }
  private Dispatcher disp;
  public synchronized void setDispatcher (Dispatcher d) {
    disp = d;
    notify();
    setActive();
  }
  private synchronized void waitForWork () throws InterruptedException {
    while (disp == null)
      wait();
  }
  private synchronized void doneWorking () {
    disp = null;
    setInactive();
  }
  public void run () {
    while (true) { // non terminating threads
      try { waitForWork(); } catch (InterruptedException e) { return; } // termination
      File f;
      while ((f = disp.nextFile()) != null)
        sendFile(f);
      doneWorking();
    }
  }
}
```

```
private static class Dispatcher {
  private int next;
  private File[] files;

  public Dispatcher (File[] files) {
    this.files = files;
  }
  public synchronized File nextFile () {
    if (next == files.length)
      return null;
    return files[next++];
  }
}

public void sendAllFilesPoolWait (File[] files, int k) {
  Dispatcher d = new Dispatcher(files);
  for (int i=0, l=Math.min(k, workers.length); i<l; i++)
    workers[i].setDispatcher(d);
  synchronized (this) {
    while (activeCount() > 0)
      try { wait(); } catch (InterruptedException e) { return; }
  }
}
```

```
class Executor {
  public void execute (Runnable task, Runnable callback) { ... }
  ...
}
```

- **tasks** (runnables) are submitted to an **executor**
- they start to run **immediately** in a **new thread**
- the call to `execute` is **asynchronous** and returns immediately
- when a task finishes, a call to the **callback object** is made

Task example:

```
class WaitTask implements Runnable {
  private final long duration;
  public WaitTask (long d) { duration = d; }
  public void run () {
    try {
      print("WaitTask("+duration+") starts");
      Thread.sleep(duration);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    } finally {
      print("WaitTask("+duration+") ends");
    }
  }
}
```

Test:

```
Executor exec = new Executor();
Runnable task = new WaitTask(5000);
Runnable callback = new Runnable() {
    public void run () {
      print("callback called");
    }
  };
print("submitting task and its callback");
exec.execute(task, callback);
print("task submitted");
print("main thread done");
```

Output:

```
10:44:23.810: submitting task and its callback
10:44:23.831: task submitted
10:44:23.832: main thread done
10:44:23.832: WaitTask(5000) starts
10:44:28.833: WaitTask(5000) ends
10:44:28.834: callback called
```

Solution:

```
public void execute (final Runnable task, final Runnable callback) {
  Thread worker = new Thread() {
      public void run () {
        task.run();
        callback.run();
      }
    };
  worker.start();
}
```

```
class Executor {
  public void execute (Runnable task, Runnable callback) { ... }
  public void execute (Runnable task, Runnable callback, Runnable error) { ... }
  ...
}
```

- same as before if the task runs **successfully**
- if the task **fails**, the `error` object is called instead (the `callback` object is not called)

Test:

```
Executor exec = new Executor();
Runnable task = new WaitTask(-1); // this task will fail
Runnable callback = new Runnable() { /* same as before */ };
Runnable error = new Runnable() {
    public void run () {
      print("something went wrong!");
    }
  };
print("submitting task, callback and error handler");
exec.execute(task, callback, error);
print("task submitted");
print("main thread done");
```

Output:

```
10:52:38.436: submitting task, callback and error handler
10:52:38.456: task submitted
10:52:38.457: main thread done
10:52:38.457: WaitTask(-1) starts
10:52:38.457: WaitTask(-1) ends
10:52:38.458: something went wrong!
```

Test:

```
Executor exec = new Executor();
Runnable task = new WaitTask(-1); // this task will fail
Runnable callback = new Runnable() { /* same as before */ };
Runnable error = new Runnable() {
    public void run () {
      print("something went wrong!");
    }
  };
print("submitting task, callback and error handler");
exec.execute(task, callback, error);
print("task submitted");
print("main thread done");
```

Output:

```
10:52:38.436: submitting task, callback and error handler
10:52:38.456: task submitted
10:52:38.457: main thread done
10:52:38.457: WaitTask(-1) starts
10:52:38.457: WaitTask(-1) ends
10:52:38.458: something went wrong!
```

Note: `Thread.sleep(-1)` throws `IllegalArgumentException`

Solution:

```
public void execute (final Runnable task,
                     final Runnable callback,
                     final Runnable error) {
  Thread worker = new Thread() {
      public void run () {
        try {
          task.run();
        } catch (Throwable t) {
          error.run();
          return;
        }
        callback.run();
      }
    };
  worker.start();
}
```

```
class Executor {
  public void execute (Runnable task, Runnable callback) { ... }
  public void execute (Runnable task, Runnable callback, Runnable error) { ... }
  public SimpleFuture execute (Runnable task) { ... }
  ...
}
```

```
interface SimpleFuture {
  boolean isDone ();
  void waitUntilDone () throws InterruptedException;
}
```

- tasks start **asynchronously** in **new threads** as before
- `execute` returns a **future** `f`
- `f.isDone()` returns *true* after the task finishes and *false* until then
- `f.waitUntilDone()` **blocks** the calling thread until the task finishes

Test:

```
Executor exec = new Executor();
Runnable task = new WaitTask(10000);
print("submitting task and getting a future");
SimpleFuture future = exec.execute(task);
print("task submitted");
Thread.sleep(5000);
print("task done: " + future.isDone());
print("main thread calling waitUntilDone");
future.waitUntilDone();
print("main thread done");
```

Output:

```
11:20:22.120: submitting task and getting a future
11:20:22.141: task submitted
11:20:22.141: WaitTask(10000) starts
11:20:27.142: task done: false
11:20:27.142: main thread calling waitUntilDone
11:20:32.142: WaitTask(10000) ends
11:20:32.142: main thread done
```

Solution:

```
public SimpleFuture execute (Runnable task) {
  FutureTask future = new FutureTask(task);
  Thread worker = new Thread(future);
  worker.start();
  return future;
}
```

- `FutureTask` is both a `Runnable` *and* a `SimpleFuture`
- when it finishes (as a runnable), its `isDone` method returns *true* and its `waitUntilDone` unblocks
- it uses `wait/notify(All)` as a mechanism to block/unblock threads

```
class FutureTask implements SimpleFuture, Runnable {
  private Runnable task;
  public FutureTask (Runnable task) {
    setTask(task);
  }
  public void run () {
    try {
      getTask().run();
    } finally {
      setTask(null);
    }
  }
  private synchronized Runnable getTask () {
    return task;
  }
  private synchronized void setTask (Runnable r) {
    task = r;
    if (r == null) notifyAll();
  }
  public synchronized boolean isDone () {
    return task == null;
  }
  public synchronized void waitUntilDone ()
  throws InterruptedException {
    while (task != null) wait();
  }
}
```

```
class FutureTask implements SimpleFuture, Runnable {
  private Runnable task;
  public FutureTask (Runnable task) {
    setTask(task);
  }
  public void run () {
    try {
      getTask().run();
    } finally {
      setTask(null);
    }
  }
  private synchronized Runnable getTask () {
    return task;
  }
  private synchronized void setTask (Runnable r) {
    task = r;
    if (r == null) notifyAll();
  }
  public synchronized boolean isDone () {
    return task == null;
  }
  public synchronized void waitUntilDone ()
  throws InterruptedException {
    while (task != null) wait();
  }
}
```

- field `task` is **always** (reading or writing) accessed within a **synchronized** block, including in the **constructor**
- the `run` method of the task is invoked **outside** the synchronized blocks (otherwise, `isDone` could not be called by other threads)
- it is usually good practice to **release locks** before a long computation (task running, I/O, . . .)

Output?

```
class A {
    String name;
    public String toString () { return name; }
    public A (String s) { name = s; }
}

class Main {
    public static void main (String[] args) {
        A a = new A("foo") {
                String name = "bar";
            };
        System.out.println(a + " " + a.name);
    }
}
```

Output?

```
class A {
    String name;
    public String toString () { return name; }
    public A (String s) { name = s; }
}

class Main {
    public static void main (String[] args) {
        A a = new A("foo") {
                String name = "bar";
                public String toString () {
                    return name;
                }
            };
        System.out.println(a + " " + a.name);
    }
}
```

What if the modifier is defined as private, final or private final?

```
class A {
    /* modifier */ int[] x = new int[10];
    public int[] m () {
        return x;
    }
}
```

```
class B {
    B () {
        new A().m()[3] = 42;
    }
}
```

Any problem?

Consider the following method, which takes a list of cars as its input, prints the prices of all the cars under $10,000, and removes cars over $9,999 from the list:

```
void printCheapCars (List l) {
   for (int i=0; i<l.size(); i++) {
     if (((Car)l.get(i)).price() >= 10000)
       l.remove(i);
     else
       System.out.print(((Car)l.get(i)).price()+" ");
   }
   System.out.println();
 }
```

Suppose a list `l` contains the following cars (in that order), displayed with their price:

`[Car(9680), Car(11979), Car(1794), Car(9908), Car(17205), Car(4591), Car(12690), Car(11398), Car(926), Car(3972)]`

Rewrite the method;

```
void printCheapCars (List<? extends Car> l)
```