

# CS 230

## 01 - Threading Introduction

Radu Nicolescu  
Department of Computer Science  
University of Auckland

15 May 2013

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Resources

For a complementary view:

- The Java Tutorial Trail: Essential Classes
  - Lesson: Concurrency
- The Java Tutorial Trail: Creating a GUI With JFC/Swing
  - Lesson: Concurrency in Swing
    - Initial Threads, The Event Dispatch Thread
- *MultiThreaded toolkits: A failed dream?*, Graham Hamilton  
[http://weblogs.java.net/blog/kggh/archive/2004/10/multithreaded\\_t.html](http://weblogs.java.net/blog/kggh/archive/2004/10/multithreaded_t.html)
- The Java API (J2SE7 Documentation)

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka "GUI" thread): runs the event loop

## Quiz: How many threads does your program have?

- Briefly, a **thread** is a single sequential flow of control within a program.
- To find out (1) the number of logical cores and (2) which thread runs a particular method, use

```
1 int Runtime.getRuntime().availableProcessors()  
2 long Thread.currentThread().getId()
```

- Command-line
  - **Initial thread**: the thread that starts the main()
- Swing
  - **Initial thread**: the thread that starts the main() and typically disappears soon thereafter
  - **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Typical problems that threading solves

- Designing **efficient** applications
  - Which use all the available processing power, e.g., on multi-core machines
  - DEMO: Ray Tracing
- Designing **responsive** applications
  - Which remain responsive even when a GUI event handler starts a long computation (or a network/database operation)
  - DEMO: Celsius Converter
- Designing programs that must do several **distinct logical tasks**
  - Browsers that must download several pages in parallel
  - Servers that must service multiple clients concurrently
  - DEMO: Dining Philosophers

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads**
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Concurrency with Threads

In general, we consider:

- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we’ll talk more about this later)

# Concurrency with Threads

In general, we consider:

- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we’ll talk more about this later)

# Concurrency with Threads

In general, we consider:

- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we’ll talk more about this later)

# Concurrency with Threads

In general, we consider:

- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we’ll talk more about this later)

# Concurrency with Threads

In general, we consider:

- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we’ll talk more about this later)

# Concurrency with Threads

In general, we consider:

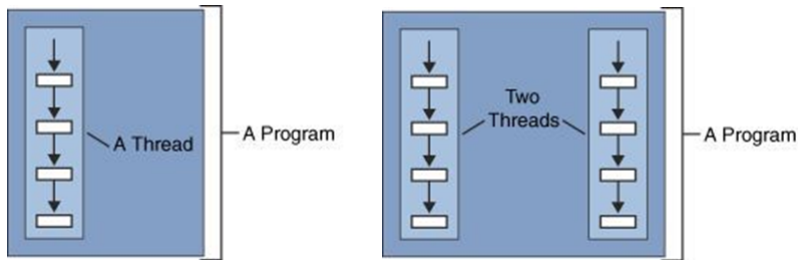
- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we'll talk more about this later)

# Concurrency with Threads

In general, we consider:

- Concurrency between processes (distinct programs inside the operating system)
- **Concurrency between threads** (“lightweight processes”, inside the same program)
- Definition: Logically, a thread is a single sequential flow of control within a program.
- Until now, we have probably used to think that a program has a single sequential flow
- However, this seldom the case, a modern program usually has quite a few threads
- Especially OO (such as Java, .NET) and GUI systems (we’ll talk more about this later)

# Threads



- Left: A single thread running within a program
- Right: Two threads running within a program

# Multiple threads within the same program

How do multiple threads physically run in the same program?  
 (abbreviations: C#=core no., T#=thread no., S#=slice no.)

- Ideally, each thread runs on its **own** core – on computers with several cores, but this is seldom possible

C#1: T#1 ...

C#2: T#2 ...

- Threads can also run **interleaved**, even on a single core, where each threads gets in turn a succession of time slices

C#1: T#1.S#1 T#2.S#1 T#2.S#2 T#3.S#2 ...

C#2: T#3.S#1 T#1.S#2 T#1.S#3 T#2.S#3 ...

# Multiple threads within the same program

How do multiple threads physically run in the same program?  
 (abbreviations: C#=core no., T#=thread no., S#=slice no.)

- Ideally, each thread runs on its **own** core – on computers with several cores, but this is seldom possible

C#1: T#1 ...

C#2: T#2 ...

- Threads can also run **interleaved**, even on a single core, where each threads gets in turn a succession of time slices

C#1: T#1.S#1 T#2.S#1 T#2.S#2 T#3.S#2 ...

C#2: T#3.S#1 T#1.S#2 T#1.S#3 T#2.S#3 ...

# Multiple threads within the same program

How do multiple threads physically run in the same program?  
(abbreviations: C#=core no., T#=thread no., S#=slice no.)

- Ideally, each thread runs on its **own** core – on computers with several cores, but this is seldom possible

C#1: T#1 ...

C#2: T#2 ...

- Threads can also run **interleaved**, even on a single core, where each threads gets in turn a succession of time slices

C#1: **T#1.S#1** T#2.S#1 T#2.S#2 T#3.S#2 ...

C#2: T#3.S#1 **T#1.S#2** **T#1.S#3** T#2.S#3 ...

## Multiple threads within the same program

- In general, the order and the size of the time slices is **not predictable**: it is risky mistake to assume anything less than *total chaos*
- The design and coding must take extra steps to ensure a proper **coordination**, if this is important

## Multiple threads within the same program

- In general, the order and the size of the time slices is **not predictable**: it is risky mistake to assume anything less than *total chaos*
- The design and coding must take extra steps to ensure a proper **coordination**, if this is important

# Multiple threads within the same program

A thread can temporarily cease its execution:

- Voluntarily: the thread itself issues **sleep** or **yield** commands – this is known as **cooperative scheduling**: *imagine a voluntary zip on lane merging*
  - DEMO: Bouncer: “selfish” threads are bad, for the system and in the end for themselves
- Enforced: by the OS scheduler – on systems that support so-called **preemptive time slicing**: *imagine lane merging with traffic lights*

# Multiple threads within the same program

A thread can temporarily cease its execution:

- Voluntarily: the thread itself issues **sleep** or **yield** commands – this is known as **cooperative scheduling**: *imagine a voluntary zip on lane merging*
  - DEMO: Bouncer: “selfish” threads are bad, for the system and in the end for themselves
- Enforced: by the OS scheduler – on systems that support so-called **preemptive time slicing**: *imagine lane merging with traffic lights*

# Multiple threads within the same program

A thread can temporarily cease its execution:

- Voluntarily: the thread itself issues **sleep** or **yield** commands – this is known as **cooperative scheduling**: *imagine a voluntary zip on lane merging*
  - DEMO: Bouncer: “selfish” threads are bad, for the system and in the end for themselves
- Enforced: by the OS scheduler – on systems that support so-called **preemptive time slicing**: *imagine lane merging with traffic lights*

# Multiple threads within the same program

A thread can temporarily cease its execution:

- Voluntarily: the thread itself issues **sleep** or **yield** commands – this is known as **cooperative scheduling**: *imagine a voluntary zip on lane merging*
  - DEMO: Bouncer: “selfish” threads are bad, for the system and in the end for themselves
- Enforced: by the OS scheduler – on systems that support so-called **preemptive time slicing**: *imagine lane merging with traffic lights*

# How Are Threads Reified?

- At a low level, **threads** (i.e. sequential statement *flows*) are typically **reified** into manipulable instances of class **Thread**
- The two faces of the thread:
  - ① Thread = a logical control flow
  - ② Thread = a “control panel” to manipulate threads (1)
- A class **Thread** is available in most modern languages, e.g., in Java, C#, ...
- OS Threads vs Java Threads : not always 1:1 (not further discussed here)

# How Are Threads Reified?

- At a low level, **threads** (i.e. sequential statement *flows*) are typically **reified** into manipulable instances of class **Thread**
- The two faces of the thread:
  - 1 Thread = a logical control flow
  - 2 Thread = a “control panel” to manipulate threads (1)
- A class **Thread** is available in most modern languages, e.g., in Java, C#, ...
- OS Threads vs Java Threads : not always 1:1 (not further discussed here)

# How Are Threads Reified?

- At a low level, **threads** (i.e. sequential statement *flows*) are typically **reified** into manipulable instances of class **Thread**
- The two faces of the thread:
  - 1 Thread = a logical control flow
  - 2 Thread = a “control panel” to manipulate threads (1)
- A class **Thread** is available in most modern languages, e.g., in Java, C#, ...
- OS Threads vs Java Threads : not always 1:1 (not further discussed here)

# How Are Threads Reified?

- At a low level, **threads** (i.e. sequential statement *flows*) are typically **reified** into manipulable instances of class **Thread**
- The two faces of the thread:
  - 1 Thread = a logical control flow
  - 2 Thread = a “control panel” to manipulate threads (1)
- A class **Thread** is available in most modern languages, e.g., in Java, C#, ...
- OS Threads vs Java Threads : not always 1:1 (not further discussed here)

# How To Use Threads—Bird's eye view

- Implement your own threads using the system **Thread** or **Runnable** types
  - Low-level and a bit costly (thread creations are costly operations, even a waiting thread takes up some resources, despite it seems doing “nothing”)
- Use thread-pools such as provide by **Executor** or **Swing**
  - Good enough for many applications, typically pools of about 25 threads, which are recycled, to avoid the cost of thread creation and destruction.
- Implicitly, using specific API such as **Timer** or the updated high-level **concurrency** framework of **Java 7**, aka **Fork-Join**

# How To Use Threads—Bird's eye view

- Implement your own threads using the system **Thread** or **Runnable** types
  - Low-level and a bit costly (thread creations are costly operations, even a waiting thread takes up some resources, despite it seems doing “nothing”)
- Use thread-pools such as provide by **Executor** or **Swing**
  - Good enough for many applications, typically pools of about 25 threads, which are recycled, to avoid the cost of thread creation and destruction.
- Implicitly, using specific API such as **Timer** or the updated high-level **concurrency** framework of **Java 7**, aka **Fork-Join**

# How To Use Threads—Bird's eye view

- Implement your own threads using the system **Thread** or **Runnable** types
  - Low-level and a bit costly (thread creations are costly operations, even a waiting thread takes up some resources, despite it seems doing “nothing”)
- Use thread-pools such as provide by **Executor** or **Swing**
  - Good enough for many applications, typically pools of about 25 threads, which are recycled, to avoid the cost of thread creation and destruction.
- Implicitly, using specific API such as **Timer** or the updated high-level **concurrency** framework of **Java 7**, aka **Fork-Join**

# How To Use Threads—Bird's eye view

- Implement your own threads using the system **Thread** or **Runnable** types
  - Low-level and a bit costly (thread creations are costly operations, even a waiting thread takes up some resources, despite it seems doing “nothing”)
- Use thread-pools such as provide by **Executor** or **Swing**
  - Good enough for many applications, typically pools of about 25 threads, which are recycled, to avoid the cost of thread creation and destruction.
- Implicitly, using specific API such as **Timer** or the updated high-level **concurrency** framework of **Java 7**, aka **Fork-Join**

# How To Use Threads—Bird's eye view

- Implement your own threads using the system **Thread** or **Runnable** types
  - Low-level and a bit costly (thread creations are costly operations, even a waiting thread takes up some resources, despite it seems doing “nothing”)
- Use thread-pools such as provide by **Executor** or **Swing**
  - Good enough for many applications, typically pools of about 25 threads, which are recycled, to avoid the cost of thread creation and destruction.
- Implicitly, using specific API such as **Timer** or the updated high-level **concurrency** framework of **Java 7**, aka **Fork-Join**

# How To Use Threads—Bird's eye view

- Implement your own threads using the system **Thread** or **Runnable** types
  - Low-level and a bit costly (thread creations are costly operations, even a waiting thread takes up some resources, despite it seems doing “nothing”)
- Use thread-pools such as provide by **Executor** or **Swing**
  - Good enough for many applications, typically pools of about 25 threads, which are recycled, to avoid the cost of thread creation and destruction.
- Implicitly, using specific API such as **Timer** or the updated high-level **concurrency** framework of **Java 7**, aka **Fork-Join**

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads**
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

## How to implement your own custom Thread subclass

```
1 public class SimpleThread extends Thread {  
2     public SimpleThread(String name) {  
3         super(name);  
4     }  
5     @Override public void run() {  
6         ...  
7     }  
8 }
```

- Subclass `Thread` and override its empty `run` method, so that it does something
- The constructor sets the Thread's name, which could be used later, e.g., via `getName()`.
- The overridden `run` method: the *heart* of any Thread — also known as its `Task`

# How to implement your own custom Thread subclass

```
1 public class SimpleThread extends Thread {  
2     public SimpleThread(String name) {  
3         super(name);  
4     }  
5     @Override public void run() {  
6         ...  
7     }  
8 }
```

- Subclass **Thread** and override its empty **run** method, so that it does something
- The constructor sets the Thread's name, which could be used later, e.g., via `getName()`.
- The overridden **run** method: the *heart* of any Thread — also known as its **Task**

# How to implement your own custom Thread subclass

```
1 public class SimpleThread extends Thread {  
2     public SimpleThread(String name) {  
3         super(name);  
4     }  
5     @Override public void run() {  
6         ...  
7     }  
8 }
```

- Subclass **Thread** and override its empty **run** method, so that it does something
- The constructor sets the Thread's name, which could be used later, e.g., via **getName()**.
- The overridden **run** method: the *heart* of any Thread — also known as its **Task**

# How to implement your own custom Thread subclass

```
1 public class SimpleThread extends Thread {  
2     public SimpleThread(String name) {  
3         super(name);  
4     }  
5     @Override public void run() {  
6         ...  
7     }  
8 }
```

- Subclass **Thread** and override its empty **run** method, so that it does something
- The constructor sets the Thread's name, which could be used later, e.g., via **getName()**.
- The overridden **run** method: the *heart* of any Thread — also known as its **Task**

## The SimpleThread's Task

```
1      @Override public void run() {
2          for (int i = 0; i < 10; i++) {
3              System.out.println(i+" "+getName());
4              try {
5                  sleep((long)(Math.random() * 1000));
6              } catch (InterruptedException ex) {}
7          }
8          System.out.println("DONE! "+getName());
9      }
```

- Here, this task *iterates* 10 times
- *prints* some results
- *sleeps* for a random time interval
- we need a **try** block, because a *sleeping* thread can be awoken by an exception triggered from another thread (*later*)

## The SimpleThread's Task

```
1      @Override public void run() {  
2          for (int i = 0; i < 10; i++) {  
3              System.out.println(i+" "+getName());  
4              try {  
5                  sleep((long)(Math.random() * 1000));  
6              } catch (InterruptedException ex) {}  
7          }  
8          System.out.println("DONE! "+getName());  
9      }
```

- Here, this task *iterates* 10 times
- *prints* some results
- *sleeps* for a random time interval
- we need a **try** block, because a *sleeping* thread can be awoken by an exception triggered from another thread (*later*)

# The SimpleThread's Task

```
1      @Override public void run() {
2          for (int i = 0; i < 10; i++) {
3              System.out.println(i+" "+getName());
4              try {
5                  sleep((long)(Math.random() * 1000));
6              } catch (InterruptedException ex) {}
7          }
8          System.out.println("DONE! "+getName());
9      }
```

- Here, this task *iterates* 10 times
- *prints* some results
- *sleeps* for a random time interval
- we need a **try** block, because a *sleeping* thread can be awoken by an exception triggered from another thread (*later*)

# The SimpleThread's Task

```
1      @Override public void run() {
2          for (int i = 0; i < 10; i++) {
3              System.out.println(i+" "+getName());
4              try {
5                  sleep((long)(Math.random() * 1000));
6              } catch (InterruptedException ex) {}
7          }
8          System.out.println("DONE! "+getName());
9      }
```

- Here, this task *iterates* 10 times
- *prints* some results
- *sleeps* for a random time interval
- we need a **try** block, because a *sleeping* thread can be awoken by an exception triggered from another thread (*later*)

# The SimpleThread's Task

```
1      @Override public void run() {
2          for (int i = 0; i < 10; i++) {
3              System.out.println(i+" "+getName());
4              try {
5                  sleep((long)(Math.random() * 1000));
6              } catch (InterruptedException ex) {}
7          }
8          System.out.println("DONE! "+getName());
9      }
```

- Here, this task *iterates* 10 times
- *prints* some results
- *sleeps* for a random time interval
- we need a **try** block, because a *sleeping* thread can be awoken by an exception triggered from another thread (*later*)

# Testing our custom SimpleThread

```
1 public class TwoThreadsTest {
2     public static void main (String [] args) {
3         new SimpleThread(" Jamaica" ).start ();
4         new SimpleThread(" Fiji" ).start ();
5
6         // something is missing here (more a few slides later)
7     }
8 }
```

- The main method creates and then starts two threads (immediately following their construction)
- by calling their `start` method
- which in turn calls our SimpleThread's `run` method
- You do **not** call `run` directly; if you do so, it will be a normal method call (no threading) – more *later*

# Testing our custom SimpleThread

```
1 public class TwoThreadsTest {
2     public static void main (String [] args) {
3         new SimpleThread(" Jamaica" ).start ();
4         new SimpleThread(" Fiji" ).start ();
5
6         // something is missing here (more a few slides later)
7     }
8 }
```

- The main method creates and then starts two threads (immediately following their construction)
- by calling their `start` method
- which in turn calls our `SimpleThread`'s `run` method
- You do **not** call `run` directly; if you do so, it will be a normal method call (no threading) – more *later*

# Testing our custom SimpleThread

```
1 public class TwoThreadsTest {
2     public static void main (String [] args) {
3         new SimpleThread(" Jamaica" ). start ();
4         new SimpleThread(" Fiji" ). start ();
5
6         // something is missing here (more a few slides later)
7     }
8 }
```

- The main method creates and then starts two threads (immediately following their construction)
- by calling their **start** method
- which in turn calls our SimpleThread's **run** method
- You do **not** call **run** directly; if you do so, it will be a normal method call (no threading) – more *later*

# Testing our custom SimpleThread

```
1 public class TwoThreadsTest {
2     public static void main (String [] args) {
3         new SimpleThread(" Jamaica" ).start ();
4         new SimpleThread(" Fiji" ).start ();
5
6         // something is missing here (more a few slides later)
7     }
8 }
```

- The main method creates and then starts two threads (immediately following their construction)
- by calling their **start** method
- which in turn calls our SimpleThread's **run** method
- You do **not** call **run** directly; if you do so, it will be a normal method call (no threading) – more *later*

# Testing our custom SimpleThread

```
1 public class TwoThreadsTest {
2     public static void main (String [] args) {
3         new SimpleThread(" Jamaica" ).start ();
4         new SimpleThread(" Fiji" ).start ();
5
6         // something is missing here (more a few slides later)
7     }
8 }
```

- The main method creates and then starts two threads (immediately following their construction)
- by calling their **start** method
- which in turn calls our SimpleThread's **run** method
- You do **not** call **run** directly; if you do so, it will be a normal method call (no threading) – more *later*

## SimpleThread's sample output (many possible) ↓

0 Jamaica	2 Fiji	7 Jamaica	DONE!
0 Fiji	3 Fiji	8 Jamaica	Jamaica
1 Jamaica	4 Jamaica	9 Jamaica	8 Fiji
1 Fiji	5 Jamaica	5 Fiji	9 Fiji
2 Jamaica	6 Jamaica	6 Fiji	DONE! Fiji
3 Jamaica	4 Fiji	7 Fiji	

- The output from each thread is intermingled with the output from the other
- The order is **not** predictable
- Worst things could happen if the two threads would attempt to use a shared data item without proper synchronization (more *later*)

## SimpleThread's sample output (many possible) ↓

0 Jamaica	2 Fiji	7 Jamaica	DONE!
0 Fiji	3 Fiji	8 Jamaica	Jamaica
1 Jamaica	4 Jamaica	9 Jamaica	8 Fiji
1 Fiji	5 Jamaica	5 Fiji	9 Fiji
2 Jamaica	6 Jamaica	6 Fiji	DONE! Fiji
3 Jamaica	4 Fiji	7 Fiji	

- The output from each thread is intermingled with the output from the other
- The order is **not** predictable
- Worst things could happen if the two threads would attempt to use a shared data item without proper synchronization (more *later*)

## SimpleThread's sample output (many possible) ↓

0 Jamaica	2 Fiji	7 Jamaica	DONE!
0 Fiji	3 Fiji	8 Jamaica	Jamaica
1 Jamaica	4 Jamaica	9 Jamaica	8 Fiji
1 Fiji	5 Jamaica	5 Fiji	9 Fiji
2 Jamaica	6 Jamaica	6 Fiji	DONE! Fiji
3 Jamaica	4 Fiji	7 Fiji	

- The output from each thread is intermingled with the output from the other
- The order is **not** predictable
- Worst things could happen if the two threads would attempt to use a shared data item without proper synchronization (more *later*)

## SimpleThread's sample output (many possible) ↓

0 Jamaica	2 Fiji	7 Jamaica	DONE!
0 Fiji	3 Fiji	8 Jamaica	Jamaica
1 Jamaica	4 Jamaica	9 Jamaica	8 Fiji
1 Fiji	5 Jamaica	5 Fiji	9 Fiji
2 Jamaica	6 Jamaica	6 Fiji	DONE! Fiji
3 Jamaica	4 Fiji	7 Fiji	

- The output from each thread is intermingled with the output from the other
- The order is **not** predictable
- Worst things could happen if the two threads would attempt to use a shared data item without proper synchronization (more *later*)

## How to implement your own Runnable class?

```
1 public class SimpleRunnable implements Runnable {
2     public void run() {
3         String name = Thread.currentThread().getName();
4
5         for (int i = 0; i < 10; i++) {
6             System.out.println(i+" "+name);
7             try {
8                 Thread.sleep((long)(Math.random() * 1000));
9             } catch (InterruptedException ex) {}
10        }
11
12        System.out.println("DONE! "+name);
13    }
14 }
```

- Similar to SimpleThread, but we need a longer way to pick up the thread's name

## How to implement your own Runnable class?

```
1 public class SimpleRunnable implements Runnable {
2     public void run() {
3         String name = Thread.currentThread().getName();
4
5         for (int i = 0; i < 10; i++) {
6             System.out.println(i+" "+name);
7             try {
8                 Thread.sleep((long)(Math.random() * 1000));
9             } catch (InterruptedException ex) {}
10        }
11
12        System.out.println("DONE! "+name);
13    }
14 }
```

- Similar to SimpleThread, but we need a longer way to pick up the thread's name

## Testing our custom SimpleRunnable

```
1 public class TwoRunnablesTest {
2     public static void main (String[] args) {
3         new Thread(new SimpleRunnable(), "Jamaica")
4             .start();
5         new Thread(new SimpleRunnable(), "Fiji")
6             .start();
7
8         // something is missing here (more a few slides later)
9     }
10 }
```

- Similar to TwoThreadsTest, but we use another Thread constructor, which takes a Runnable object (our SimpleRunnable's)

## Testing our custom SimpleRunnable

```
1 public class TwoRunnablesTest {
2     public static void main (String[] args) {
3         new Thread(new SimpleRunnable(), "Jamaica")
4             .start();
5         new Thread(new SimpleRunnable(), "Fiji")
6             .start();
7
8         // something is missing here (more a few slides later)
9     }
10 }
```

- Similar to TwoThreadsTest, but we use another Thread constructor, which takes a Runnable object (our SimpleRunnable's)

# When to use a Thread subclass and when a runnable task?

- Subclassing from Thread may seem *simpler...*
- However, implementing a Runnable task offers more *flexibility*
- It allows you to subclass from a *different* type (i.e., *not* from Thread)
- It provides support for more advanced scenarios, that use the Task concept

# When to use a Thread subclass and when a runnable task?

- Subclassing from Thread may seem *simpler...*
- However, implementing a Runnable task offers more *flexibility*
- It allows you to subclass from a *different* type (i.e., *not* from Thread)
- It provides support for more advanced scenarios, that use the Task concept

# When to use a Thread subclass and when a runnable task?

- Subclassing from Thread may seem *simpler...*
- However, implementing a Runnable task offers more *flexibility*
- It allows you to subclass from a *different* type (i.e., *not* from Thread)
- It provides support for more advanced scenarios, that use the Task concept

# When to use a Thread subclass and when a runnable task?

- Subclassing from Thread may seem *simpler...*
- However, implementing a Runnable task offers more *flexibility*
- It allows you to subclass from a *different* type (i.e., *not* from Thread)
- It provides support for more advanced scenarios, that use the Task concept

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads**
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Daemon Threads

- There are two kinds of threads: **daemon** threads and **non-daemon** threads
- **Daemon** threads, aka **service** or **background** threads: provide support for the non-daemon threads and don't need to exist without these
- Consequence: if all non-daemon threads terminate, then the daemon threads automatically terminate (daemon threads **cannot** keep the application alive)
- Non-daemon threads:
  - the **main thread** (for console apps)
  - the **event dispatching thread** (for GUI apps) – in most frameworks, only this thread is legally allowed to **update** the GUI (therefore, it is also known, colloquially, as the **GUI thread**)

# Daemon Threads

- There are two kinds of threads: **daemon** threads and **non-daemon** threads
- **Daemon** threads, aka **service** or **background** threads: provide support for the non-daemon threads and don't need to exist without these
- Consequence: if all non-daemon threads terminate, then the daemon threads automatically terminate (daemon threads **cannot** keep the application alive)
- Non-daemon threads:
  - the **main thread** (for console apps)
  - the **event dispatching thread** (for GUI apps) – in most frameworks, only this thread is legally allowed to **update** the GUI (therefore, it is also known, colloquially, as the **GUI thread**)

# Daemon Threads

- There are two kinds of threads: **daemon** threads and **non-daemon** threads
- **Daemon** threads, aka **service** or **background** threads: provide support for the non-daemon threads and don't need to exist without these
- Consequence: if all non-daemon threads terminate, then the daemon threads automatically terminate (daemon threads **cannot** keep the application alive)
- Non-daemon threads:
  - the **main thread** (for console apps)
  - the **event dispatching thread** (for GUI apps) – in most frameworks, only this thread is legally allowed to **update** the GUI (therefore, it is also known, colloquially, as the **GUI thread**)

# Daemon Threads

- There are two kinds of threads: **daemon** threads and **non-daemon** threads
- **Daemon** threads, aka **service** or **background** threads: provide support for the non-daemon threads and don't need to exist without these
- Consequence: if all non-daemon threads terminate, then the daemon threads automatically terminate (daemon threads **cannot** keep the application alive)
- Non-daemon threads:
  - the **main thread** (for console apps)
  - the **event dispatching thread** (for GUI apps) – in most frameworks, only this thread is legally allowed to **update** the GUI (therefore, it is also known, colloquially, as the **GUI thread**)

# Daemon Threads

- There are two kinds of threads: **daemon** threads and **non-daemon** threads
- **Daemon** threads, aka **service** or **background** threads: provide support for the non-daemon threads and don't need to exist without these
- Consequence: if all non-daemon threads terminate, then the daemon threads automatically terminate (daemon threads **cannot** keep the application alive)
- Non-daemon threads:
  - the **main thread** (for console apps)
  - the **event dispatching thread** (for GUI apps) – in most frameworks, only this thread is legally allowed to **update** the GUI (therefore, it is also known, colloquially, as the **GUI thread**)

# Daemon Threads

- There are two kinds of threads: **daemon** threads and **non-daemon** threads
- **Daemon** threads, aka **service** or **background** threads: provide support for the non-daemon threads and don't need to exist without these
- Consequence: if all non-daemon threads terminate, then the daemon threads automatically terminate (daemon threads **cannot** keep the application alive)
- Non-daemon threads:
  - the **main thread** (for console apps)
  - the **event dispatching thread** (for GUI apps) – in most frameworks, only this thread is legally allowed to **update** the GUI (therefore, it is also known, colloquially, as the **GUI thread**)

# Daemon Threads

- You can *check* if a thread is daemon by `isDaemon()`
- You can *mark* a thread as daemon by `setDaemon(true)` or
- *mark* it as non-daemon by `setDaemon(false)`

# Daemon Threads

- You can *check* if a thread is daemon by `isDaemon()`
- You can *mark* a thread as daemon by `setDaemon(true)` or
- *mark* it as non-daemon by `setDaemon(false)`

# Daemon Threads

- You can *check* if a thread is daemon by `isDaemon()`
- You can *mark* a thread as daemon by `setDaemon(true)` or
- *mark* it as non-daemon by `setDaemon(false)`

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination**
- ⑦ Thread Joins
- ⑧ Thread Interrupts
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# How to terminate an application that uses threads?

- A *single-threaded* console application terminates when its main and only thread, which is non-daemon, completes its task (e.g., reaches the final “}”)
- A *multiple-threaded* console application terminates when its main thread, which is non-daemon, and all its other non-daemon threads terminate
- These two extra lines, at the end of our demos, prompt the user to terminate the main thread, which will also terminate the program, **if** all the other threads are daemon

```
1     System.out.println("<enter> to exit");  
2     System.in.read();
```

# How to terminate an application that uses threads?

- A *single-threaded* console application terminates when its main and only thread, which is non-daemon, completes its task (e.g., reaches the final “}”)
- A *multiple-threaded* console application terminates when its main thread, which is non-daemon, and all its other non-daemon threads terminate
- These two extra lines, at the end of our demos, prompt the user to terminate the main thread, which will also terminate the program, **if** all the other threads are daemon

```
1     System.out.println("<enter> to exit");  
2     System.in.read();
```

# How to terminate an application that uses threads?

- A *single-threaded* console application terminates when its main and only thread, which is non-daemon, completes its task (e.g., reaches the final “}”)
- A *multiple-threaded* console application terminates when its main thread, which is non-daemon, and all its other non-daemon threads terminate
- These two extra lines, at the end of our demos, prompt the user to terminate the main thread, which will also terminate the program, **if** all the other threads are daemon

```
1     System.out.println("<enter> to exit");  
2     System.in.read();
```

## How to terminate an application that uses threads?

- A *single-threaded* console application terminates when its main and only thread, which is non-daemon, completes its task (e.g., reaches the final “}”)
- A *multiple-threaded* console application terminates when its main thread, which is non-daemon, and all its other non-daemon threads terminate
- These two extra lines, at the end of our demos, prompt the user to terminate the main thread, which will also terminate the program, **if** all the other threads are daemon

```
1      System.out.println("<enter> to exit");  
2      System.in.read();
```

# How to terminate an application that uses threads?

- A GUI application has a non-daemon thread running the GUI event dispatching loop
- typically, closing the main form will stop this GUI thread *and*
- *if* there is no other non-daemon thread, all daemon threads will be stopped and the application will terminate
- An application can be also stopped by explicitly calling `System.exit` ...

# How to terminate an application that uses threads?

- A GUI application has a non-daemon thread running the GUI event dispatching loop
- typically, closing the main form will stop this GUI thread *and*
- if there is no other non-daemon thread, all daemon threads will be stopped and the application will terminate
- An application can be also stopped by explicitly calling `System.exit ...`

# How to terminate an application that uses threads?

- A GUI application has a non-daemon thread running the GUI event dispatching loop
- typically, closing the main form will stop this GUI thread *and*
- **if** there is no other non-daemon thread, all daemon threads will be stopped and the application will terminate
- An application can be also stopped by explicitly calling `System.exit ...`

# How to terminate an application that uses threads?

- A GUI application has a non-daemon thread running the GUI event dispatching loop
- typically, closing the main form will stop this GUI thread *and*
- **if** there is no other non-daemon thread, all daemon threads will be stopped and the application will terminate
- An application can be also stopped by explicitly calling **System.exit** ...

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins**
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Thread Joins

To allow one thread to wait for the completion of another thread, use the method `Thread.join()`, e.g.:

```
1 Thread tj = new SimpleThread("Jamaica");
2 Thread tf = new SimpleThread("Fiji");
3
4 tj.start();
5 tf.start();
6
7 tj.join();
8 tf.join();
9
10 System.out.println("JOINED!");
```

# Thread Joins

To allow one thread to wait for the completion of another thread, use the method `Thread.join()`, e.g.:

```
1 Thread tj = new SimpleThread(" Jamaica" );
2 Thread tf = new SimpleThread(" Fiji" );
3
4 tj.start();
5 tf.start();
6
7 tj.join();
8 tf.join();
9
10 System.out.println(" JOINED!" );
```

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions**
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Thread interruptions

To interrupt another thread, use the method `Thread.interrupt()`, e.g.:

```
1 Thread tj = new SimpleThread("Jamaica");
2 tj.start();
3
4 ...
5
6 tj.interrupt();
```

This requires some cooperation from the interrupted thread, such as *tj*

# Thread interruptions

To interrupt another thread, use the method `Thread.interrupt()`, e.g.:

```
1 Thread tj = new SimpleThread("Jamaica");
2 tj.start();
3
4 ...
5
6 tj.interrupt();
```

This requires some cooperation from the interrupted thread, such as *tj*

# Thread interruptions

To interrupt another thread, use the method `Thread.interrupt()`, e.g.:

```
1 Thread tj = new SimpleThread("Jamaica");  
2 tj.start();  
3  
4 ...  
5  
6 tj.interrupt();
```

This requires some cooperation from the interrupted thread, such as *tj*

# Thread interruptions

Methods such as `Thread.sleep(...)`, `Thread.join(...)` and `Object.wait(...)` *automatically check* if the current thread has received an `interrupt()`, and, if so, will raise an `InterruptedException`

Without such methods, such a check should be periodically performed by calling `Thread.interrupted()`

```
1   if (interrupted ()) ...
```

# Thread interruptions

Methods such as `Thread.sleep(...)`, `Thread.join(...)` and `Object.wait(...)` *automatically check* if the current thread has received an `interrupt()`, and, if so, will raise an `InterruptedException`

Without such methods, such a check should be periodically performed by calling `Thread.interrupted()`

```
1  if (interrupted ()) ...
```

# Thread interruptions

Methods such as `Thread.sleep(...)`, `Thread.join(...)` and `Object.wait(...)` *automatically check* if the current thread has received an `interrupt()`, and, if so, will raise an `InterruptedException`

Without such methods, such a check should be periodically performed by calling `Thread.interrupted()`

```
1  if ( interrupted () ) ...
```

# Thread interruptions

Consequently, threads which use **Thread.sleep(...)**, **Thread.join(...)** or **Object.wait(...)** must provide adequate exception handling, e.g.:

```
1  try {
2      sleep((long)(Math.random() * 2000));
3  } catch (InterruptedException ex) {
4      System.out.println("INTERRUPTED! "+getName());
5      return;
6  }
```

# Thread interruptions

Consequently, threads which use **Thread.sleep(...)**, **Thread.join(...)** or **Object.wait(...)** must provide adequate exception handling, e.g.:

```
1  try {  
2      sleep((long)(Math.random() * 2000));  
3  } catch (InterruptedException ex) {  
4      System.out.println("INTERRUPTED! "+getName());  
5      return;  
6  }
```

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API**
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Summary of Thread class API – check Java Doc

```
1 public interface Runnable {  
2     void run()  
3 }
```

```
1 public class Thread implements Runnable {  
2     Thread()  
3     Thread(String name)  
4     Thread(Runnable target)  
5     Thread(Runnable target, String name)  
6  
7     public void start() // call this to start the run  
8     public void run() // fill this but never call it directly  
9  
10    // continued on next slide
```

# Summary of Thread class API – check Java Doc

```
1 public interface Runnable {  
2     void run()  
3 }
```

```
1 public class Thread implements Runnable {  
2     Thread()  
3     Thread(String name)  
4     Thread(Runnable target)  
5     Thread(Runnable target, String name)  
6  
7     public void start() // call this to start the run  
8     public void run() // fill this but never call it directly  
9  
10    // continued on next slide
```

# Summary of Thread class API – check Java Doc

```
1 public class Thread implements Runnable {  
2     // continued from previous slide  
3  
4     public static Thread currentThread()  
5     public String getName()  
6     public boolean isDaemon()  
7     public void setDaemon(boolean on)  
8     public void join() throws InterruptedException  
9     public void join(long millis)  
10                    throws InterruptedException  
11     public void sleep(long millis)  
12                    throws InterruptedException  
13     public void interrupt()  
14     public static boolean interrupted()  
15     public static void yield()  
16 }
```

# Summary of Thread class API – check Java Doc

```
1 public class Thread implements Runnable {  
2     // continued from previous slide  
3  
4     public static Thread currentThread()  
5     public String getName()  
6     public boolean isDaemon()  
7     public void setDaemon(boolean on)  
8     public void join() throws InterruptedException  
9     public void join(long millis)  
10         throws InterruptedException  
11     public void sleep(long millis)  
12         throws InterruptedException  
13     public void interrupt()  
14     public static boolean interrupted()  
15     public static void yield()  
16 }
```

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference**
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Thread interference

Consider an object of the following class, **concurrently** used by several threads:

```
1 class Counter {  
2     private int c = 0;  
3     public int value() { return c; }  
4     public void increment() { c++; }  
5     public void decrement() { c--; }  
6 }
```

What can go wrong?

# Thread interference

Consider an object of the following class, **concurrently** used by several threads:

```
1 class Counter {  
2     private int c = 0;  
3     public int value() { return c; }  
4     public void increment() { c++; }  
5     public void decrement() { c--; }  
6 }
```

What can go wrong?

# Thread interference

Consider an object of the following class, **concurrently** used by several threads:

```
1 class Counter {  
2     private int c = 0;  
3     public int value() { return c; }  
4     public void increment() { c++; }  
5     public void decrement() { c--; }  
6 }
```

What can go wrong?

# Thread interference

- Imagine that two threads call `increment()`, *successively*. If initially `c==0`, then the result should and will be `c==2`.
- Imagine now that two threads call `increment()`, at about the *same time*. What can go wrong?
- Even a simple operation such as `c++` is *not atomic*; it corresponds to a sequence of atomic operations such as the following (where `r` is one the thread's arithmetic registers):

```
1 1. LOAD  r , c  // LOAD is a read operation
2 2. INC   r      // INC is a modify operation
3 3. STORE r , c  // STORE is a write operation
```

- What can go wrong?

# Thread interference

- Imagine that two threads call `increment()`, *successively*. If initially `c==0`, then the result should and will be `c==2`.
- Imagine now that two threads call `increment()`, at about the *same time*. What can go wrong?
- Even a simple operation such as `c++` is *not atomic*; it corresponds to a sequence of atomic operations such as the following (where `r` is one the thread's arithmetic registers):

```
1 1. LOAD  r , c    // LOAD is a read operation
2 2. INC   r        // INC is a modify operation
3 3. STORE r , c    // STORE is a write operation
```

- What can go wrong?

# Thread interference

- Imagine that two threads call `increment()`, *successively*. If initially `c==0`, then the result should and will be `c==2`.
- Imagine now that two threads call `increment()`, at about the *same time*. What can go wrong?
- Even a simple operation such as `c++` is *not atomic*; it corresponds to a sequence of atomic operations such as the following (where `r` is one the thread's arithmetic registers):

```
1 1. LOAD  r , c    // LOAD is a read operation
2 2. INC   r        // INC is a modify operation
3 3. STORE r , c    // STORE is a write operation
```

- What can go wrong?

# Thread interference

- Imagine that two threads call `increment()`, *successively*. If initially `c==0`, then the result should and will be `c==2`.
- Imagine now that two threads call `increment()`, at about the *same time*. What can go wrong?
- Even a simple operation such as `c++` is *not atomic*; it corresponds to a sequence of atomic operations such as the following (where `r` is one the thread's arithmetic registers):

```
1 1. LOAD  r , c  // LOAD is a read operation
2 2. INC   r      // INC is a modify operation
3 3. STORE r , c  // STORE is a write operation
```

- What can go wrong?

# Thread interference

- Imagine that two threads call `increment()`, *successively*. If initially `c==0`, then the result should and will be `c==2`.
- Imagine now that two threads call `increment()`, at about the *same time*. What can go wrong?
- Even a simple operation such as `c++` is *not atomic*; it corresponds to a sequence of atomic operations such as the following (where `r` is one the thread's arithmetic registers):

1	1. LOAD	<code>r, c</code>	// LOAD is a read operation
2	2. INC	<code>r</code>	// INC is a modify operation
3	3. STORE	<code>r, c</code>	// STORE is a write operation

- What can go wrong?

# Thread interference

- The two threads can interleave in many ways, some of them with unexpected wrong results:

```

1  Thread t1
2  .
3  LOAD r1 , c
4  INC r1
5  .
6  .
7  .
8  STORE r1 , c
  
```

```

1  Thread t2
2  .
3  .
4  .
5  LOAD r2 , c
6  INC r2
7  STORE r2 , c
8  .
  
```

- In the above case, the final value of both `r1` and `r2` will be `1`!
- This value `1` will be stored twice in `c`, with the net wrong result `c==1`, instead of the expected `2`!

# Thread interference

- The two threads can interleave in many ways, some of them with unexpected wrong results:

```

1  Thread t1
2  .
3  LOAD r1 , c
4  INC r1
5  .
6  .
7  .
8  STORE r1 , c
  
```

```

1  Thread t2
2  .
3  .
4  .
5  LOAD r2 , c
6  INC r2
7  STORE r2 , c
8  .
  
```

- In the above case, the final value of both `r1` and `r2` will be `1`!
- This value `1` will be stored twice in `c`, with the net wrong result `c==1`, instead of the expected `2`!

# Thread interference

- The two threads can interleave in many ways, some of them with unexpected wrong results:

```

1  Thread t1
2  .
3  LOAD r1 , c
4  INC r1
5  .
6  .
7  .
8  STORE r1 , c
  
```

```

1  Thread t2
2  .
3  .
4  .
5  LOAD r2 , c
6  INC r2
7  STORE r2 , c
8  .
  
```

- In the above case, the final value of both **r1** and **r2** will be **1**!
- This value **1** will be stored twice in **c**, with the net wrong result **c==1**, instead of the expected **2**!

# Thread interference

- The two threads can interleave in many ways, some of them with unexpected wrong results:

```

1  Thread t1
2  .
3  LOAD r1 , c
4  INC r1
5  .
6  .
7  .
8  STORE r1 , c
  
```

```

1  Thread t2
2  .
3  .
4  .
5  LOAD r2 , c
6  INC r2
7  STORE r2 , c
8  .
  
```

- In the above case, the final value of both **r1** and **r2** will be **1**!
- This value **1** will be stored twice in **c**, with the net wrong result **c==1**, instead of the expected **2**!

# Atomic operations

- A *single* **read** or a **write** for **reference** variables and most primitive values: **int**, **float**, but not **long** and **double**.
- A *single* **read** or a **write** for variables declared **volatile** (including **long** and **double**)
- Problem: How to protect non-atomic operations or longer operation sequences (e.g., read+modify+write) against interference?
- We'll see this over the next lectures

# Atomic operations

- A *single* **read** or a **write** for **reference** variables and most primitive values: **int**, **float**, but not **long** and **double**.
- A *single* **read** or a **write** for variables declared **volatile** (including **long** and **double**)
- Problem: How to protect non-atomic operations or longer operation sequences (e.g., read+modify+write) against interference?
- We'll see this over the next lectures

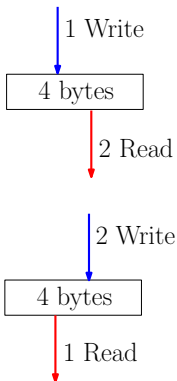
# Atomic operations

- A *single* **read** or a **write** for **reference** variables and most primitive values: **int**, **float**, but not **long** and **double**.
- A *single* **read** or a **write** for variables declared **volatile** (including **long** and **double**)
- Problem: How to protect non-atomic operations or longer operation sequences (e.g., read+modify+write) against interference?
- We'll see this over the next lectures

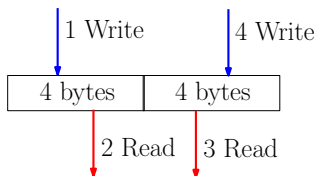
# Atomic operations

- A *single* **read** or a **write** for **reference** variables and most primitive values: **int**, **float**, but not **long** and **double**.
- A *single* **read** or a **write** for variables declared **volatile** (including **long** and **double**)
- Problem: How to protect non-atomic operations or longer operation sequences (e.g., read+modify+write) against interference?
- We'll see this over the next lectures

# Left: `int` R/W atomic $\checkmark$ ; Right: `long` R/W ?



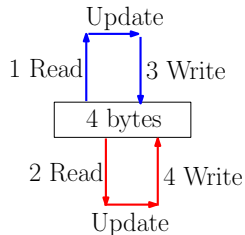
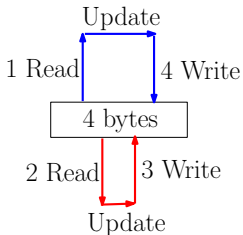
As for read, but inconsistent scenarios are also possible, e.g.:



You can fix this, by declaring that **long** variable **volatile**

Note that **volatile** does *not* help in more complex scenarios, e.g., if each thread executes a read+update+write operation (*next slide*)

# int R/U/W scenarios with inconsistent results



Making the **int** variable **volatile** does *not* help here

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency**
- ⑫ Swing and Threads

# Memory consistency errors

Assume that  $x$  and  $y$  start with the initial value 0.

```

1  Thread t1
2  STORE 1, x // x = 1;
3  ...
4  STORE 2, y // y = 2;
  
```

```

1  Thread t2
2  LOAD r2, y // ... = y;
3  ...
4  LOAD r2, x // ... = x;
  
```

In “pathological” cases, thread  $t2$  may see  $y==2$ , but still  $x==0$ !

Causes for memory consistency errors:

- Compilers attempt optimizations which may *reorder* existing statements:

$$\boxed{x = 1; y = 2;} \Rightarrow \boxed{y = 2; x = 1;}$$

- next slide*

# Memory consistency errors

Assume that  $x$  and  $y$  start with the initial value 0.

```

1 Thread t1
2 STORE 1, x // x = 1;
3 ...
4 STORE 2, y // y = 2;
  
```

```

1 Thread t2
2 LOAD r2, y // ... = y;
3 ...
4 LOAD r2, x // ... = x;
  
```

In “pathological” cases, thread  $t2$  may see  $y==2$ , but still  $x==0$ !

Causes for memory consistency errors:

- Compilers attempt optimizations which may *reorder* existing statements:

```

x = 1; y = 2; ⇒ y = 2; x = 1;
  
```

- next slide*

# Memory consistency errors

Assume that  $x$  and  $y$  start with the initial value 0.

```

1  Thread t1
2  STORE 1, x // x = 1;
3  ...
4  STORE 2, y // y = 2;
  
```

```

1  Thread t2
2  LOAD r2, y // ... = y;
3  ...
4  LOAD r2, x // ... = x;
  
```

In “pathological” cases, thread  $t2$  may see  $y==2$ , but still  $x==0$ !

Causes for memory consistency errors:

- Compilers attempt optimizations which may *reorder* existing statements:

```

x = 1; y = 2;  ⇒  y = 2; x = 1;
  
```

- next slide*

# Memory consistency errors

Assume that  $x$  and  $y$  start with the initial value 0.

```

1  Thread t1
2  STORE 1, x // x = 1;
3  ...
4  STORE 2, y // y = 2;
  
```

```

1  Thread t2
2  LOAD r2, y // ... = y;
3  ...
4  LOAD r2, x // ... = x;
  
```

In “pathological” cases, thread  $t2$  may see  $y==2$ , but still  $x==0$ !

Causes for memory consistency errors:

- Compilers attempt optimizations which may *reorder* existing statements:

```

x = 1; y = 2; ⇒ y = 2; x = 1;
  
```

- next slide*

# Memory consistency errors

Assume that  $x$  and  $y$  start with the initial value 0.

```

1  Thread t1
2  STORE 1, x // x = 1;
3  ...
4  STORE 2, y // y = 2;
  
```

```

1  Thread t2
2  LOAD r2, y // ... = y;
3  ...
4  LOAD r2, x // ... = x;
  
```

In “pathological” cases, thread  $t2$  may see  $y==2$ , but still  $x==0$ !

Causes for memory consistency errors:

- Compilers attempt optimizations which may *reorder* existing statements:

```

x = 1; y = 2; ⇒ y = 2; x = 1;
  
```

- *next slide*

# Memory consistency errors

Assume that  $x$  and  $y$  start with the initial value 0.

```

1  Thread t1
2  STORE 1, x // x = 1;
3  ...
4  STORE 2, y // y = 2;
  
```

```

1  Thread t2
2  LOAD r2, y // ... = y;
3  ...
4  LOAD r2, x // ... = x;
  
```

In “pathological” cases, thread  $t2$  may see  $y==2$ , but still  $x==0$ !

Causes for memory consistency errors:

- Compilers attempt optimizations which may *reorder* existing statements:

$$\boxed{x = 1; y = 2;} \Rightarrow \boxed{y = 2; x = 1;}$$

- *next slide*

# Memory consistency errors–Not required

Sample scenario where reordering may occur:

```
1 Source code  
2 x = ...  
3 y = ...  
4 z = ... y ...
```

```
1 Reordered code  
2 y = ...  
3 x = ...  
4 z = ... y ...
```

Variable *y* is evaluated first, before *x*, to make pipelined evaluation of *z* more likely (here given by an expression which uses *y*, but not *x*)

# Memory consistency errors–Not required

Sample scenario where reordering may occur:

```
1 Source code  
2 x = ...  
3 y = ...  
4 z = ... y ...
```

```
1 Reordered code  
2 y = ...  
3 x = ...  
4 z = ... y ...
```

Variable *y* is evaluated first, before *x*, to make pipelined evaluation of *z* more likely (here given by an expression which uses *y*, but not *x*)

## Memory consistency errors–Not required

Sample scenario where reordering may occur:

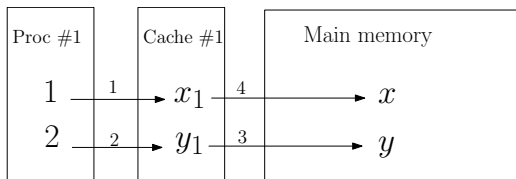
```
1 Source code  
2 x = ...  
3 y = ...  
4 z = ... y ...
```

```
1 Reordered code  
2 y = ...  
3 x = ...  
4 z = ... y ...
```

Variable *y* is evaluated first, before *x*, to make pipelined evaluation of *z* more likely (here given by an expression which uses *y*, but not *x*)

# Memory consistency errors—Not required

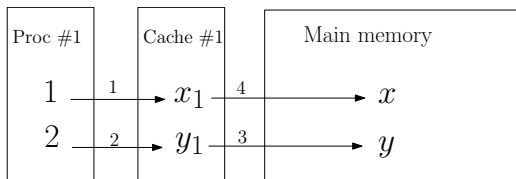
Another cause for memory consistency errors: cache consistency model (architecture dependent)



- Besides main memory, copies of the variables are kept in cache memories or registers — thus the same variable may have *multiple copies*, not always synchronized (simplified diagram)

# Memory consistency errors—Not required

Another cause for memory consistency errors: cache consistency model (architecture dependent)



- Besides main memory, copies of the variables are kept in cache memories or registers — thus the same variable may have *multiple copies*, not always synchronized (simplified diagram)

## Memory consistency errors–Not required

- Memory consistency is treated differently by different Java versions, using different *happens-before* and *memory barrier* models.
- It also depends on the *physical architecture* of the machine.
- Such errors are avoided by the *synchronization* and *lock* techniques which will be taught (if properly applied).
- We do **not** further this topic in this course.

## Memory consistency errors–Not required

- Memory consistency is treated differently by different Java versions, using different *happens-before* and *memory barrier* models.
- It also depends on the *physical architecture* of the machine.
- Such errors are avoided by the *synchronization* and *lock* techniques which will be taught (if properly applied).
- We do **not** further this topic in this course.

## Memory consistency errors–Not required

- Memory consistency is treated differently by different Java versions, using different *happens-before* and *memory barrier* models.
- It also depends on the *physical architecture* of the machine.
- Such errors are avoided by the *synchronization* and *lock* techniques which will be taught (if properly applied).
- We do **not** further this topic in this course.

## Memory consistency errors–Not required

- Memory consistency is treated differently by different Java versions, using different *happens-before* and *memory barrier* models.
- It also depends on the *physical architecture* of the machine.
- Such errors are avoided by the *synchronization* and *lock* techniques which will be taught (if properly applied).
- We do **not** further this topic in this course.

# Outline

- ① Resources
- ② Overview
- ③ Concurrency with Threads
- ④ Simple Custom Threads
- ⑤ Daemon Threads
- ⑥ Threads Termination
- ⑦ Thread Joins
- ⑧ Thread Interruptions
- ⑨ Thread API
- ⑩ Thread Interference
- ⑪ Memory Consistency
- ⑫ Swing and Threads

# Single Threaded GUIs

As mentioned in a previous handout, Swing is **NOT thread-safe**: there are important restrictions concerning GUI access from other threads

- Most Swing component methods (create, update, repaint, ...) must be invoked from the GUI (event dispatch) thread **only!**
- There are a few exceptions, mentioned in the API as “**thread safe**”
- Programs that ignore these rules may appear to function correctly, most of the time, but are subject to unpredictable errors
- This is true, not only for Swing, but for all modern GUI toolkits, including .NET

# Single Threaded GUIs

As mentioned in a previous handout, Swing is **NOT thread-safe**: there are important restrictions concerning GUI access from other threads

- Most Swing component methods (create, update, repaint, ...) must be invoked from the GUI (event dispatch) thread **only!**
- There are a few exceptions, mentioned in the API as “**thread safe**”
- Programs that ignore these rules may appear to function correctly, most of the time, but are subject to unpredictable errors
- This is true, not only for Swing, but for all modern GUI toolkits, including .NET

# Single Threaded GUIs

As mentioned in a previous handout, Swing is **NOT thread-safe**: there are important restrictions concerning GUI access from other threads

- Most Swing component methods (create, update, repaint, ...) must be invoked from the GUI (event dispatch) thread **only!**
- There a few exceptions, mentioned in the API as “**thread safe**”
- Programs that ignore these rules may appear to function correctly, most of the time, but are subject to unpredictable errors
- This is true, not only for Swing, but for all modern GUI toolkits, including .NET

# Single Threaded GUIs

As mentioned in a previous handout, Swing is **NOT thread-safe**: there are important restrictions concerning GUI access from other threads

- Most Swing component methods (create, update, repaint, ...) must be invoked from the GUI (event dispatch) thread **only!**
- There a few exceptions, mentioned in the API as **“thread safe”**
- Programs that ignore these rules may appear to function correctly, most of the time, but are subject to unpredictable errors
- This is true, not only for Swing, but for all modern GUI toolkits, including .NET

# Single Threaded GUIs

As mentioned in a previous handout, Swing is **NOT thread-safe**: there are important restrictions concerning GUI access from other threads

- Most Swing component methods (create, update, repaint, ...) must be invoked from the GUI (event dispatch) thread **only!**
- There a few exceptions, mentioned in the API as “**thread safe**”
- Programs that ignore these rules may appear to function correctly, most of the time, but are subject to unpredictable errors
- This is true, not only for Swing, but for all modern GUI toolkits, including .NET

## Failed dreams — thought-provoking article

Graham Hamilton, *Multithreaded toolkits: A failed dream?*

*There are certain ideas in Computer Science that I think of as the “Failed Dreams” (borrowing a term from Vernor Vinge).*

*The Failed Dreams seem like obvious good ideas. So they get periodically reinvented, and people put a lot of time and thought into them.*

*They typically work well on a research scale and they have the intriguing attribute of almost working on a production scale. Except you can never quite get all the kinks ironed out...*

*For me, multithreaded GUI toolkits seem to be one of the Failed Dreams. ...*

## Failed dreams — thought-provoking article

Graham Hamilton, *Multithreaded toolkits: A failed dream?*

*There are certain ideas in Computer Science that I think of as the “Failed Dreams” (borrowing a term from Vernor Vinge).*

*The Failed Dreams seem like obvious good ideas. So they get periodically reinvented, and people put a lot of time and thought into them.*

*They typically work well on a research scale and they have the intriguing attribute of almost working on a production scale. Except you can never quite get all the kinks ironed out...*

*For me, multithreaded GUI toolkits seem to be one of the Failed Dreams. ...*

## Failed dreams — thought-provoking article

Graham Hamilton, *Multithreaded toolkits: A failed dream?*

*There are certain ideas in Computer Science that I think of as the “Failed Dreams” (borrowing a term from Vernor Vinge).*

*The Failed Dreams seem like obvious good ideas. So they get periodically reinvented, and people put a lot of time and thought into them.*

*They typically work well on a research scale and they have the intriguing attribute of almost working on a production scale. Except you can never quite get all the kinks ironed out...*

*For me, multithreaded GUI toolkits seem to be one of the Failed Dreams. ...*

## Failed dreams — thought-provoking article

Graham Hamilton, *Multithreaded toolkits: A failed dream?*

*There are certain ideas in Computer Science that I think of as the “Failed Dreams” (borrowing a term from Vernor Vinge).*

*The Failed Dreams seem like obvious good ideas. So they get periodically reinvented, and people put a lot of time and thought into them.*

*They typically work well on a research scale and they have the intriguing attribute of almost working on a production scale. Except you can never quite get all the kinks ironed out...*

*For me, multithreaded GUI toolkits seem to be one of the Failed Dreams. ...*

## Failed dreams — thought-provoking article

Graham Hamilton, *Multithreaded toolkits: A failed dream?*

*There are certain ideas in Computer Science that I think of as the “Failed Dreams” (borrowing a term from Vernor Vinge).*

*The Failed Dreams seem like obvious good ideas. So they get periodically reinvented, and people put a lot of time and thought into them.*

*They typically work well on a research scale and they have the intriguing attribute of almost working on a production scale. Except you can never quite get all the kinks ironed out...*

*For me, multithreaded GUI toolkits seem to be one of the Failed Dreams. ...*

# Swing specific threads

- **Initial thread:** the thread that starts the `main()`
- **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**, executes drawing and short event handling code - **should not** run time-consuming tasks
- **Worker (background) threads:** for time consuming tasks - **MUST NOT** interact directly with the GUI (few exceptions)
- Two *high-level* solutions for the proper interaction with the GUI thread
  - Using **SwingUtilities**: here
  - Using **SwingWorker**: in the last course section (optionally)

# Swing specific threads

- **Initial thread**: the thread that starts the `main()`
- **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**, executes drawing and short event handling code - **should not** run time-consuming tasks
- **Worker (background) threads**: for time consuming tasks - **MUST NOT** interact directly with the GUI (few exceptions)
- Two *high-level* solutions for the proper interaction with the GUI thread
  - Using **SwingUtilities**: here
  - Using **SwingWorker**: in the last course section (optionally)

# Swing specific threads

- **Initial thread**: the thread that starts the `main()`
- **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**, executes drawing and short event handling code - **should not** run time-consuming tasks
- **Worker (background) threads**: for time consuming tasks - **MUST NOT** interact directly with the GUI (few exceptions)
- Two *high-level* solutions for the proper interaction with the GUI thread
  - Using **SwingUtilities**: here
  - Using **SwingWorker**: in the last course section (optionally)

# Swing specific threads

- **Initial thread**: the thread that starts the main()
- **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**, executes drawing and short event handling code - **should not** run time-consuming tasks
- **Worker (background) threads**: for time consuming tasks - **MUST NOT** interact directly with the GUI (few exceptions)
- Two *high-level* solutions for the proper interaction with the GUI thread
  - Using **SwingUtilities**: here
  - Using **SwingWorker**: in the last course section (optionally)

# Swing specific threads

- **Initial thread**: the thread that starts the main()
- **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**, executes drawing and short event handling code - **should not** run time-consuming tasks
- **Worker (background) threads**: for time consuming tasks - **MUST NOT** interact directly with the GUI (few exceptions)
- Two *high-level* solutions for the proper interaction with the GUI thread
  - Using **SwingUtilities**: here
  - Using **SwingWorker**: in the last course section (optionally)

# Swing specific threads

- **Initial thread**: the thread that starts the `main()`
- **Event-dispatch thread** (EDT, aka “GUI” thread): runs the **event loop**, executes drawing and short event handling code - **should not** run time-consuming tasks
- **Worker (background) threads**: for time consuming tasks - **MUST NOT** interact directly with the GUI (few exceptions)
- Two *high-level* solutions for the proper interaction with the GUI thread
  - Using **SwingUtilities**: here
  - Using **SwingWorker**: in the last course section (optionally)

## SwingUtilities.invokeLater(*runnable-task*)

- usually invoked by the initial thread, to safely create and start the GUI
- also, often invoked from a worker thread
- schedules a given task to be **asynchronously** executed by the event-dispatch thread and returns immediately (without waiting the actual execution)

```
1 SwingUtilities.invokeLater(new Runnable() {
2     public void run() {
3         // this code will be executed by the event-dispatch thread,
4         // e.g., to update the GUI
5     }
6 });
```

## SwingUtilities.invokeLater(*runnable-task*)

- usually invoked by the initial thread, to safely create and start the GUI
- also, often invoked from a worker thread
- schedules a given task to be **asynchronously** executed by the event-dispatch thread and returns immediately (without waiting the actual execution)

```
1 SwingUtilities.invokeLater(new Runnable() {  
2     public void run() {  
3         // this code will be executed by the event-dispatch thread,  
4         // e.g., to update the GUI  
5     }  
6 });
```

## SwingUtilities.invokeLater(*runnable-task*)

- usually invoked by the initial thread, to safely create and start the GUI
- also, often invoked from a worker thread
- schedules a given task to be **asynchronously** executed by the event-dispatch thread and returns immediately (without waiting the actual execution)

```
1 SwingUtilities.invokeLater(new Runnable() {  
2     public void run() {  
3         // this code will be executed by the event-dispatch thread,  
4         // e.g., to update the GUI  
5     }  
6 });
```

## SwingUtilities.invokeLater(*runnable-task*)

- usually invoked by the initial thread, to safely create and start the GUI
- also, often invoked from a worker thread
- schedules a given task to be **asynchronously** executed by the event-dispatch thread and returns immediately (without waiting the actual execution)

```
1 SwingUtilities.invokeLater(new Runnable() {
2     public void run() {
3         // this code will be executed by the event-dispatch thread,
4         // e.g., to update the GUI
5     }
6 });
```

## SwingUtilities.invokeLater(*runnable-task*)

- usually invoked by the initial thread, to safely create and start the GUI
- also, often invoked from a worker thread
- schedules a given task to be **asynchronously** executed by the event-dispatch thread and returns immediately (without waiting the actual execution)

```
1 SwingUtilities.invokeLater(new Runnable() {  
2     public void run() {  
3         // this code will be executed by the event-dispatch thread,  
4         // e.g., to update the GUI  
5     }  
6 });
```

## SwingUtilities.invokeLaterAndWait(*runnable-task*)

- invokeAndWait() is similar to invokeLater(), but **synchronously**, i.e., the calling thread waits until the event-dispatch thread completes the given task

## SwingUtilities.invokeLaterAndWait(*runnable-task*)

- invokeAndWait() is similar to invokeLater(), but **synchronously**, i.e., the calling thread waits until the event-dispatch thread completes the given task