Software Construction

Hand Book

COMSATS INSTITUTE OF INFORMATION TECHNOLOGY

(Virtual Campus)
Islamabad, Pakistan
Lecture No. 1
Introduction to Software Construction

1.1 Subject Overview
In earlier courses, you learned how to write programs to solve small problems. In this course we teach programming “in the large”. Large software systems have many stakeholders. What will its users want? Can we describe user requirements, accurately and succinctly? Large software systems are very complex. Can we describe the design of a complex software system, accurately and succinctly? Can we be sure that a complex system will do what it is designed to do, and that it will not do anything unintended?

In this course you will learn some incomplete answers to these difficult questions. I will also attempt to teach you how to “learn how to learn” the technical skills you will need in the future – as a competent computer professional.

1.2 Main Parts of Syllabus

Four Themes
The object-oriented programming paradigm
– Object-orientation, object-oriented programming concepts and programming language constructs – because, for many important problems, OO design is a convenient way to express the problem and its solution in software.

Frameworks
– Inversion of control, AWT/Swing and JUnit – because many important “sub-problems” have already been solved: these solutions should be re-used!

Software quality
– Testing, inspection, documentation are used because large teams are designing, implementing, debugging, maintaining, revising, and supporting complex software.
Application-level concurrent programming

– Multithreading concepts, language primitives and abstractions will be studied because even our laptops have multiple CPUs. Dual-core smartphones are now available as well.

1.3 Introduction to OO Design

A process of determining what the stakeholders require, designing a set of classes with objects which will meet these requirements, implementing, and delivering.
– Structure Diagrams: high-level design
– Use Cases: high-level requirements
– Other concerns: implementation, quality assurance

1.3.1 Software Design

Communication:
Identify stakeholders, find out what they want and need.

Planning:
List tasks, identify risks, obtain resources, define milestones, estimate schedule.

Modeling:
Develop structure diagrams and use cases, maybe some other UML artifacts.

Construction:
Implement the software, with assured quality.

Deployment:
Deliver the software, and then get feedback for possible revision.

1.3.2 What is Object-Oriented Design?

– In OO design, a system is a collection of interacting objects. Each object should have simple attributes and behaviors. Each object should have simple relations to other objects.

– In procedural design, a system is a collection of basic blocks. Each basic block should have a simple effect on local and global variables. Basic blocks are linked by control-flow arcs: if/then/else, call/return, while/loop, for/loop, case, goto.
– In data architecture, a system is a collection of data structures, with access and update methods. Each data structure should have simple relations to other data structures.

Figure 1: Object Oriented Design

1.3.3 What is an Object?

Object is a building block for OO development. Like various objects in the world around us. Objects have state and behaviour

**Examples:**

**Dog**

– State/field/attribute: name, colour, isHungry, …
– Behaviour: bark(), fetch(), eat(), …

**Bicycle**

– State: gear, colour, …
– Behaviour: brake(), turn(), changeGear(), …

**VCR**

– State: brand, colour, isOn …
– Behaviour: play(), stop(), rewind(), turnOn(), …

1.4 Classes & Objects

Class
– A set of objects with shared behavior and individual state
– Individual state: Data is stored with each instance, as an instance variable.
– Shared behavior: Code is stored with the class object, as a method.
– Shared state may be stored with the class object, as a class variable.

Object
– Objects are created from classes at runtime by instantiation usually with New.
– There may be zero, one, or many objects (instances) of a class.
– Instantiated objects are garbage-collected if no other user-defined object can reference them.

1.5 Imagine a world of communicating objects

Object
An object remembers things (i.e. it has a memory): its state. An object responds to messages it gets from other objects.
– It performs the method with the given parameters, and then sends a response.
– An object that receives a strange message may throw an exception. Be careful!
An object’s method may “ask for help” from other objects.
– It sends a message to an object, and waits for a response.
– A method may send a message to itself! This is called recursion. Be careful.

Messages between objects
Usually: method calls and method returns, sometimes exceptions.

1.6 Information Hiding

The implementation details of a method should be of no concern to the sender of the message.
– If a JavaKid tells a JavaDog to fetch(), the dog might run across a busy street during its fetch().
– Parameterised methods allow the senders to have more control over object behaviour. For
example, a JavaDog might have a parameterised fetch() method:

- ball = dog.fetch(SAFELY);

**Example 1: Ball**

- Attributes
  Represent the internal state of an instance of this class.
- Constructor
  Creates the object
- Methods
  Implement the processing performed by or to an object, often updating its state.

If there are read and write methods for an attribute x, these should be called getX() and setX().

```java
public class Ball {
    public final static int SIZE = 20;
    private int xPos;
    private int yPos;
    private Color color;
    public Ball(int x, int y, Color c) {
        xPos = x;
        yPos = y;
        color = c;
    }
    public void move(int deltaX, int deltaY) {
        xPos += deltaX;
        yPos += deltaY;
    }
    public void paint(Graphics g) {
        g.setColor(color);
        g.fillOval(xPos,yPos,SIZE,SIZE);
    }
}
```
Object Instantiation

- When a constructor method is called, a new instance is created.

```java
Ball b = new Ball(10, 20, Color.Red);
Ball c = new Ball(0, 10, Color.Blue);
```

- If a class definition doesn’t include a constructor method, the Java compiler inserts a default constructor with default initialisations.

```java
public class Class1 {
    private int x;
    // Note no explicit constructor
    public int increment() {
        ++x;
    }
}
```

// is this good code?

Lecture No. 2

Introduction to Object-Orientation

2.1 Message Passing

In a method call, a message is passed to a receiver object. The receiver’s response to the message is determined by its class.

```java
Ball b = new Ball(10, 20, Color.Red);
```

```java
public class Ball {
    ... public void move(int deltaX, int deltaY) {
        xPos += deltaX;
        yPos += deltaY;
    }
}
```
2.2 Instance & Class Variables

Class variables are statically allocated, so they are shared by an entire Class of objects. The runtime system allocates class variables once per class, regardless of the number of instances created of that class. Static storage is allocated when the class is loaded. All instances share the same copy of the class variables.

Instance variables are dynamically allocated, so they may have different values in each instance of an object. When an object is instantiated, the runtime system allocates some memory to this instance – so that it can “remember” the values it stores in instance variables.

2.3 Instance & Class Methods

Instance methods operate on this object's instance variables. They also have read & write access to class variables.

Class methods are static. Class methods cannot access instance variables. Class methods are handled by the “class object” – they can be called even if there are no instances of this class.

Example:

Class1App

```java
public class Class1App {
    public static void main( String[] args ) {
        Class1 x = new Class1();
        System.out.println(
            "Without initialisation, ++x = 
            + x.increment()
        );
        System.out.println(
            "After another incrementation, ++x = 
            + x.increment()
        );
    }
```
public class BallApp extends Frame {
    Ball b = new Ball(20, 30, Color.blue);
    public BallApp() {
        addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            }
        );
        setSize(300, 200);
        setVisible(true);
    }
    public void paint(Graphics g) {
        b.paint(g);
    }
    public static void main(String[] args) {
        new BallApp();
    }
}
2.4 Unified Modeling Language (UML)

When creating complex OO systems, where do we start? When building complex systems, it might be worthwhile to plan things out before you start coding! When building a house, we usually have a set of plans.

UML is a language which allows us to graphically model an OO system in a standardised format.

- This helps us (and others!) understand the system.

There are many different UML diagrams, allowing us to model designs from many different viewpoints. Roughly, there are

- Structure diagrams (documenting the architecture), e.g. class diagrams
- Behaviour diagrams (documenting the functionality), e.g. use-case diagrams

2.5 Object Diagrams in UML

An object diagram is a graphic representation of an instance model, showing the state of a system after some objects have been instantiated, and after some variables of these objects have been updated. Object diagrams are very helpful in tuition, but are not commonly used outside the classroom.

Please focus on the basics.

- Understand the distinction between static variables and instance variables.
- Develop a working understanding of instantiation – this is a crucial concept!
- Learn how to draw UML-standard class diagrams.
- Honours-level students might want to learn more about object diagrams

2.6 Tool Support: Eclipse & ArgoUML?

You may use any development environment for this class.

Eclipse

- The de-facto industry standard for Java developers. It’s FOSS: free and open-source software. Its codebase is robust and is under active development. Your tutors will help you learn this.

DrJava

- Learning resources are available in COMPSCI 101 & 105 webareas.
javac and a customisable text editor such as emacs
I reckon every Java developer should know how to run javac from a console, but I won’t attempt to teach this!

You’ll also have to draw some class diagrams and use-case diagrams. Options:

By hand: during exams and tests, and maybe your best choice for homework

ArgoUML

FOSS, works ok (with some missing features e.g. no “undo” button – save your versions carefully!!)
No longer under active development: v0.34 is dated 15 December 2011.
No integration with current versions of Eclipse.
I don’t know of any FOSS, nor of any inexpensive software package, which does a better job of forward- and reverse-engineering between UML and Java, and which supports both class and use-case diagrams.

A general-purpose drawing package

Warning: you’ll have trouble with the fancy arrowheads in UML!

Important Points

The OO approach is based on modeling the real world using interacting objects.

OO design is a process of determining what the stakeholders require, designing a set of classes with objects which will meet these requirements, implementing, and delivering.

The statements in a class define what its objects remember and what they can do (the messages they can understand), that is, they define

Instance variables, class variables, instance methods, and class methods

The hardest concept in this set of lecture slides: instantiation.

Very important!

A UML class diagram shows the “bare bones” of an OO system design.

It need not show all classes! (A diagram should not have irrelevant information.)
Lecture No. 3
Use Cases

3.1 Introduction to Use Case modelling

What? A process of determining what the stakeholders require – by decomposing their requirements into tasks (or “use cases”) for each class of stakeholders.

How? Stakeholder Identification, Requirements Elicitation, Use Case Diagrams

Why learn this? Use cases are widely used in the industry, because they seem to work pretty well, aren’t very expensive to develop, and are at a good level of detail for end-users.

Major alternatives (not taught in this course): user stories (for agile development), formal specifications (for safety-critical software).

Software Design (Review)

Communication:
Identify stakeholders, find out what they want and need.

Planning:
List tasks, identify risks, obtain resources, define milestones, estimate schedule.

Modeling:
Develop structure diagrams and use cases, maybe some other UML artifacts.

Construction:
Implement the software, with assured quality.

Deployment:
Deliver the software, and then get feedback for possible revision.

3.2 Stakeholder Identification

Identify a variety of stakeholders, by asking yourself:

– Who is likely to be affected by, or to have an effect on, this system?

Classify the stakeholders you know about. Anyone who will directly use the system is a
stakeholder. Anyone who will be indirectly affected (in a major way) is a stakeholder.

– Anyone who pays for, or otherwise controls the design of the system is a stakeholder.

–Advertisers are important stakeholders for Google’s online search service.

Governments are stakeholders, if their laws constrain the design of a system (e.g. because citizens could be greatly harmed by the system).

Note: use cases depict the requirements of direct stakeholders (users), but you’ll have to use another method (e.g. natural language) to describe the requirements of indirect and external stakeholders.

– Reflect on your classification – have you missed an important class?

### 3.3 Use Case Analysis

To start developing use cases, ask yourself:

– What useful tasks could be performed by my system, upon request by a user?

– You probably won’t “get it right” at first. (It’ll never be perfect, but could be improved…)

To validate your current set of use cases, talk to stakeholders! Ask them “Would you use a system, if it would help you do …?” If they start telling you how they want the system to handle a use-case, then you have validated this use-case.

– You should record their detailed requirements, in natural language, as notes which accompany your use case.

– If their detailed requirements are infeasible or contradictory, you should take careful note of this!

If they tell you about some other task they’d like the system to help them with, you should document this as a possible use-case.

– Your system can’t do everything!

– Whenever you discover that you can’t deliver on all use cases within your current resources, you should communicate with your stakeholders to negotiate a feasible set.

### 3.4 An Example: Video System

John’s Video Store is an Information System which supports the following business functions:

– Recording information about videos the store owns

– This database is searchable by staff and all customers
– Information about which customer is renting which videos
– Access by staff, and also by customers who is asking about themselves
– Staffs are able to record video rentals and returns by customers.
– John doesn’t trust his customers to make these entries in their own records!
– Staff can maintain customer, video and staff information.
– Privacy requirements: customers cannot access information about other customers, personal information about customers must be accurate and relevant to John’s Video Store
– Managers of the store can generate various reports.

Who are the stakeholders?
– John’s
– Staff
– Customers
– Managers

Who are the tasks?
– Recording information about videos the store owns
– This database is searchable by staff and all customers
– Information about which customer is renting which videos
– Access by staff, and also by customers who is asking about themselves
– Staffs are able to record video rentals and returns by customers.
– Staff can maintain customer, video and staff information.
– Managers of the store can generate various reports.

3.5 Requirements Documentation

Use case descriptions
– A brief statement of what happens during each use case.
– The previous slide is a good start on this, but it’s not well-organised

Use case diagrams show
– Stick-figure actors, interacting with the system (a box)
– Choose easily-understood names for your classes of stakeholders!
John’s Video Store might have three actors: Customer, Staff, and Manager.

(Hmmm… is John an actor? Does he have a special use-case which is so important that we must add it to our diagram? Hold this question…)

Ovals (“use cases”) within the box, with easily-understood names, e.g. “Rent a video”.

Lines (“associations”) between actors and ovals

Optionally: arrowheads, extension cases, included cases, subsystems.

Example: John’s Video Store

John’s Video Store, with HR module
3.6 Video System – Designing the Classes

In this system, information stored includes:

- Videos - unique ID; title; category (children's, drama, comedy, etc); cost per night to rent; number of copies video store has available; rating
- Staff - unique ID; name; password; position
- Customers - unique ID; name; password; address; phone #
- Rentals - date rented, customer who rent the video and whether video returned

Functions this system provides include:

- Staff can add, update, delete and find videos
- Staff can add, update, delete and find people.
- Staff can rent out videos to customer and indicate videos have been returned.
- Various reporting functions e.g. number of videos rented this month are provided for managers.

3.7 Use Case Descriptions

Use case descriptions should be detailed enough that system analysts can design the classes (by grouping attributes and decomposing functions), and determine the non-functional requirements:

- “What the system should be” (or always be doing), as distinguished from “what the system should do, upon request”;
- “What the system shouldn’t do” (security requirements);
- Usability, auditability, performance, efficiency, capacity, scalability, extensibility, availability, reliability, integrity, recovery, compatibility, portability, maintainability, transparency, legal conformance

Semi-formal Use Cases

In some development environments (e.g. in the IBM Rational Unified Process), use cases are semi-formal documents with a required structure e.g.

- Title: the “goal the use case is trying to satisfy” [Fowler, 2004]
- Main Success Scenario: a numbered list of steps
- Step: “a simple statement of the interaction between the actor and a system” [Fowler, 2004]
Extensions: separately numbered lists, one per extension

To learn more, see the Wikipedia article on “Use Case”.

Example: a semi-formal use case for SearchForVideos

1. Used by Staff via an application to query for videos by title.
2. Event Flow:
   2.1 Repeat Until Exit Program
      2.1.1 Staff types in part of title in text field,
      2.1.2 Staff clicks “Search” button and a list of matching videos are returned showing ID and title. If no videos found, goto step 2.2. If error, goto step 2.3.
      2.1.3 Staff types in a ID. More information is displayed about the video e.g. rating, price to rent, etc
      2.1.4 Exit Program
   2.2 No videos found - error message displayed. Goto 2.1.1
   2.3 Database Error – error message displayed. Goto 2.1.1
   3. Related Actors and Use Cases: Staff may perform this search for a Customer. No inclusions.
      Included in Rent/Return Videos and Maintain Videos.
4. Special conditions: NONE

3.8 How to draw UML class diagrams?

Sketch by hand
Use a general-purpose graphics editor
Use ArgoUML, or some other specialised graphics editor
Ideally, your UML tool is integrated with your IDE.
   Forward engineering: document your requirements with use cases, develop your design with class diagrams, then start coding.
   Reverse engineering: inspect the code to discover its class structure and use cases.
   ArgoUML does a good job of reverse-engineering class diagrams.
   ArgoUML is clueless about reverse-engineering use cases. (Do you understand why this form of reverse-engineering is very difficult?)
Alternatives to use cases

- Story, in agile development: A one-sentence description of a feature which could be implemented quickly (i.e. tomorrow, or by the end of this week).
- Formal specification, in safety-critical development: a precise statement, in a formal language, of the post-conditions which will hold after a system action is completed, given some pre-conditions (which are also formally specified), with some accompanying, explicit, and validated assumptions about the system and its environment.

Lecture No. 4
Object-Oriented Design – Key Concepts

4.1 Abstraction

An abstraction is a view or representation of an entity that includes only the attributes of significance in a particular context. Abstraction is essential when working with complex systems. Without abstraction, the programmer faces an overwhelming level of detail.

![Figure 4.1: Abstraction](image)

4.2 Information Hiding: Two Definitions

A programming technique, in which the programmer
- uses language features (such as Interface, protected, private) to restrict access to
implementation details.

- In the extreme case, other programmers are not allowed to “look inside” your model to see its implementation.

A design technique, in which the designer

- defines a model which is simple, useful, and unlikely to change.

- In the extreme case, other programmers can read your high-level design documents, but not your code.

It is usually undesirable for programmers to rely on “undocumented functions” in an implementation.

- Undocumented functions are subject to change with every release!

It is often undesirable, and usually infeasible, to absolutely prevent other programmers from learning about your implementation.

- Many non-functional requirements (e.g. for security) are not adequately disclosed in high-level design documents.

- The design documents may be incomplete, incorrect, or ambiguous.

4.3 Inheritance and Instantiation

Inheritance: create new classes from existing classes.

- Instantiation: create new instances from existing classes.

- Inheritance is more powerful than instantiation.

- When a subclass is created by inheritance from a superclass, some of the methods and attributes in its superclass may be added or redefined.

Inheritance is an “is-a” relation.

- Example: “An orange is a fruit. An apple is a fruit.”

- Every instance of an Orange is a Fruit (= its superclass), but it is more accurately described as being an instance of the Orange class (= its subclass).

- If there is no important difference between oranges and apples, you should simplify your design!
4.4 Polymorphism

Different objects can respond differently to the same message.

– Inheritance is the “obvious” way to obtain polymorphic behaviour in your OO design, but it may not be the best way.

– Instantiations are polymorphic, if the values in their attributes affect the behaviour of their methods.

Hmmm… if you have instantiated a million objects of a single Class, could they do anything useful?

– Worker ants are (nearly) identical, but they won’t reproduce without a Queen ant.

– Ants may be important members of an ecosystem, but only if the ecosystem contains other forms of life, some inanimate objects, and an energy source.

– One way to conceive of OOD is that you’re designing an ecosystem with multiple species (Classes) in an evolutionary tree.

– It is possible to write a useful program in a non-OO language!

– Polymorphism is not necessary in programming, but it is fundamental to OO design.

4.5 Object Oriented Analysis: Basic Questions

What classes and instances should be in my high-level design?

– To get started on answering this question, you should identify important entities, and look for commonalities among different entities.

– Similar entities might be instances of the same class… but maybe there are some “natural” subclasses?

How should my classes and instances be related to each other?

– We have seen the Inheritance (“is-a”) relationship.

– We have also seen the Instantiation (“instance-of”) relationship.

– We will now look at a few other relationships that are fundamental to OO design.

4.6 Composition and Aggregation

These relationships create a complex class from one or more existing classes (or instances), in a whole/part relationship.
Composition ("owns-a", "has-a"):
- An object is a component of at most one composition.
- When a composition instance is destroyed, all objects belonging to this instance are destroyed as well.
- Example: SimCard has-a SIM

Aggregation ("has-a"):
- An object can be in many aggregations.
- Example: Pond has-a Duck

4.7 Association

In an association, an instance of one class is linked to an instance of another class.
- An aggregation is an association, because the aggregating class has instance variables which refer to objects of its parts.
- A composition is an association, because the "container" or "owner" has references to its parts.

An association may have no "container", "owner", or "whole".
- Example: every teacher has 0 or more students, and every student has 0 or 1 teachers.

4.8 Multiplicities

The multiplicity of an association may be important enough to include in a high-level design document.
- The filled-diamond notation for compositions implies that the "whole" (Composite) class has a multiplicity of 1..1 or 0..1 – because each Part can belong to at most one whole.

![Diagram](image)

- No department can exist unless it is part of a university, so the University’s multiplicity in this association is 1..1 (sometimes written “1”).
- A university must have at least one department.

Multiplicities of 0..* are not very informative.
4.9 Inheritance

Generalisation
- Look for conceptual commonalities in the abstractions
- Common attributes/state
- Common methods or behaviour

“is-a” relationship
- Subclass “is-a” kind of Superclass

Superclass (base class) defines the general state and behaviour
Subclass (derived) class:
- Created from an existing class, by extending the existing class, rather than rewriting it

Examples
- Person
- Employee & Customer (Customer “is-a” Person)

4.10 Composition Vs Inheritance

Suppose we want to create a class to represent circles:
- Composition or Inheritance?
- Circle is a point with a radius?
- Circle has a point with a radius?

General guideline
- Inheritance : Is-a
- Composition : has-a
But there is no rulebook - the objective is clean, understandable and maintainable code that can be implemented in reasonable time.

Inheritance provides a means for constructing highly reusable components, but needs to be used very carefully.

Choose composition first when creating new classes from existing classes. You should only use inheritance if it is required by your design. If you use inheritance where composition will work, your designs will become needlessly complicated.
5.1 Example of Inheritance

Too much detail?

Simplify! (with a little more notation)

Multiplicity:

- There is exactly one teacher per course, as indicated by the $1$.
- A lecturer can teach any number of courses, as indicated by $0..*$.
- We can also write $1..*$ in a UML diagram.

The arrowheads indicate that the taughtBy association is navigable in both directions, telling us
that

– Course has an instance variable teacher, of type Lecturer, and
– Lecturer has the instance variable Vector<Course> course.

Simplify even more, with defaults

Associations have default multiplicity 1. Association endpoints have a default name.

– Course has an instance variable myLecturer of type Lecturer
– Lecturer has an instance variable myCourse of type Vector<Course>

Getters, setters may be implied.

– Unimportant members might not be shown.

Defaults may be well-defined by an organisation’s stylesheet, or (more commonly) by their UML-drawing software package.

**One-way Navigation**

These courses have a Vector of their Lecturers and Students.

- (They might have a List; they might have an array; these are implementation decisions.)

These lecturers don’t know what they are teaching!

These students have no idea of what course they are taking!

**Creating new classes by generalising**

Lecturers and students have some attributes in common.

- A public name
- An email address that is revealed to everyone in our University
- A secret password

We write these methods once for the Person class, and we can reuse them in the Lecturer and
Student class.

− But we have complicated our design by adding another class.
− Do we really need so many classes?

5.2 Association Relationships

We can model objects that contain other objects by way of special associations called aggregations and compositions.

An aggregation specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

A composition indicates a strong ownership and coincident lifetime of parts by the whole (i.e., they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.
5.3 Interfaces

Interfaces are a way to specify behaviour (a public contract) without data or implementation. Interfaces are classed with an extra label next to their name: <<Interface>>

A dotted open-triangle arrow, from a class to an interface means that “the class implements this interface”.

- We also say that “the class fulfils the contract specified by this interface”, or that it “realizes the interface.”

Interface Realization Relationship

A realization relationship connects a class with an interface that supplies its behavioural specification. It is rendered by a dashed line with a hollow triangle towards the specifier.
Parameterized Class
A parameterized class or template defines a family of potential elements. To use it, the parameter must be bound.

A template is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.

*Binding* is done with the $<<\text{bind}>>$ stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.

Here we create a linked-list of names for the Dean’s List.

Enumeration
An enumeration is a user-defined data type that consists of a name and an ordered list of enumeration literals.
5.4 Packages

A package is a container-like element for organizing other elements into groups. A package can contain classes and other packages and diagrams. Packages can be used to provide controlled access between classes in different packages.

Classes in the *FrontEnd* package and classes in the *BackEnd* package cannot access each other in this diagram.
Classes in the BackEnd package now have access to the classes in the FrontEnd package.

We can model generalizations and dependencies between packages.

6.1 Interfaces, in UML

Interfaces specify behaviour (a public contract), without data or implementation. Interfaces are drawn like classes, but without attributes, and with the keyword <<Interface>>

A dotted open-triangle arrow, from a class to an interface, means that “the class implements this interface”.
– We also say that “the class fulfils the contract specified by this interface”, or that it “realizes the interface.”

### 6.2 Interfaces in Java

An Interface is like a Class, with no bodies in the methods. It may define constants (public static final) but no runtime variables.

– Usually, an Interface is public.
– An interface provides a standard way to access a class which could be implemented in many different ways.

**The Java Tutorials:**

– “There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a ‘contract’ that spells out how their software interacts.”
– “Each group should be able to write their code without any knowledge of how the other group's code is written.”
– “Generally speaking, interfaces are such contracts.”

### 6.3 Implementations, in Java

In Java, a class which implements an interface must provide an implementation of every method defined within the interface.

– A class may implement some additional methods (but these extra methods aren’t accessible through this interface)
– Beware: adding another method to an existing Interface will “break” every current implementation of this Interface!
– A class can implement many interfaces.
– An Interface can extend other Interfaces.
– Extension is the preferred way to add new methods to an Interface.
– (Do you understand why?)
– In Java, classes are less extendible than interfaces, because a Class can extend at most one other Class (“single inheritance”).
– class MountainBike extends Bicycle { … }
**Example 1**

```java
public interface Bicycle {
    void changeCadence(int newValue);
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}

class ACMEBicycle implements Bicycle {
    int cadence = 0;
    void changeCadence(int newValue) {
        cadence = newValue;
    }
    \ \ // an implementation may be incorrect!
    void changeGear(int newValue) {}  
    void speedUp(int increment) {} 
    void applyBrakes(int decrement) {} 
}
```

**Example 2**

```java
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
    // constant declarations
    // base of natural logarithms
    double E = 2.718282;
    // method signatures
    void doSomething( int i, double x );
    int doSomethingElse( String s );
}
```

**Example 3**

```java
public interface EventListener {
```
// No constants
// No method signatures!

} – “A tagging interface that all event listener interfaces must extend.”

Why?

– At first glance, this is worse than useless! One more name for the Java programmer to remember…
– This interface allows programmers, and the Java compiler, to distinguish event-listeners from all other types of classes and interfaces.
– Event-listeners are important, and they behave quite differently to a regular class. (Later, you’ll learn about inversion of control…)

6.4 MouseListener in java.awt.event

public interface MouseListener extends EventListener

The listener interface for receiving “interesting” mouse events (press, release, click, enter, and exit) on a component. (To track mouse moves and mouse drags, use the MouseMotionListener.)

All Known Subinterfaces:

MouseListener, MouseMotionListener

All Known Implementing Classes:

AWTEventMulticaster, ButtonListener, ComboBoxListener, InvocationHandler, MouseInputListener, MouseMotionListener, MouseWheelListener, ScrollListener, StandardEvent, WindowListener, ComponentListener, KeyListener, FocusListener, ActionListener, ItemListener, ChangeListener, ContainerListener

public interface MouseMotionListener extends EventListener

{ mouseDragged( MouseEvent e );
  mouseMoved( MouseEvent e );
}

public interface MouseInputListener

extends MouseListener, MouseMotionListener

{ // this interface has 7 method signatures, can you list them?
}
6.5 UML: Review of Interfaces: Packages

```
java.util

<<interface>>
+ EventListener

java.awt.event

<<interface>>
+ MouseMotionListener
+ mouseDragged()
+ mouseMoved()

<<interface>>
+ MouseListener
+ mouseClicked()
+ mouseEntered()
+ mouseExited()
+ mousePressed()
+ mouseReleased()

<<interface>>
+ MouseInputListener

<<realize>>
+ DrawingArea
```

6.6 Using an Interface as a Type

“When you define a new interface, you are defining a new reference data type.

– “You can use interface names anywhere you can use any other data type name.
– “If you define a reference variable whose type is an interface, any object you assign to
  it must be an instance of a class that implements the interface.”
– A method for finding the largest object in a pair of objects, for any objects that are
  instantiated from a class that implements Relatable

```java
public interface Relatable {
    public int isLargerThan( Relatable other );
}

public Object findMax(Object object1, Object object2) {
    Relatable obj1 = (Relatable)object1;
    Relatable obj2 = (Relatable)object2;
    if( (obj1).isLargerThan(obj2) > 0 )
        return object1;
```
If comparisons are important in your application, then you’ll be able to write very elegant code!

– You can write \( z.\text{findMax}(x, y) \), if \( x \) and \( y \) are instances of any class which extends `Relatable`.

### 6.6.1 Using an Interface as a Type: Dangers

```java
public Object findMax( Object object1, Object object2 ) {
    Relatable obj1 = (Relatable)object1;
    Relatable obj2 = (Relatable)object2;
    if( (obj1).isLargerThan(obj2) > 0 )
        return object1;
    else return object2;
}
```

– Watch out for type mismatches!! We’d get a compile-time error if
– `(object1).isLargerThan(object2)` were in the body of this method,
– we invoked it as `z.findMax(x, y)`, for any instance \( x \) of a class that doesn’t extend `Relatable`, or if
– we invoked it as `x.findLargest(y, z)`, if `y.isLargerThan()` does not accept `z` as a parameter.

If it’s easy for a programmer (and a compiler) to determine the class of all arguments in every method-call, then you’ll avoid these dangers.

### 6.7 Typing Rules

The typing rules for interfaces are similar to those for classes.

– A reference variable of interface type T can refer to an instance of any class that implements interface T or a sub-interface of T.
– Through a reference variable of interface type T, methods defined by T and its super interfaces can be called.

`instanceof`

You can use the `instanceof` operator to test an object to see if it implements an interface, before
you invoke a method in this interface.

– This might improve readability and correctness.
– This might be a hack.
– Where possible, you should extend classes and interfaces to obtain polymorphic behaviour, rather than making a runtime check.

Lecture No. 7
Java Implementation 2

7.1 Abstract Classes

Sometimes, it’s appropriate to partly-implement a class or interface. Abstract classes allow code to be reused in similar implementations. Abstract classes may include some abstract methods. If there are no abstract methods, then the class is usually (but not always) implemented fully enough to be used by an application.

Sometimes it’s helpful to have multiple implementations that differ only in their type, but this is quite an advanced concept in design.

public abstract class MyGraphicObject {
    // declare fields – these may be non-static
    private int x, y;
    // declare non-abstract methods
    // (none)
    // declare methods which must be implemented later
    abstract void draw();
}
7.1.1 Final

The final keyword can be applied to prevent the extension (over-riding) of a field, argument, method, or class.

- Final field: constant
- Final argument: cannot change the data within the called method
- Final method: cannot override method in subclasses
- Final class: cannot be subclassed (all of its methods are implicitly final as well)

7.1.2 Super

If your method overrides one of its superclass's methods, you can invoke the overridden method through the use of the keyword super.

- You can also use super to refer to a hidden field (although hiding fields is discouraged).
- Example below. (What is printed when main() is executed?)

```java
public class Superclass {
    public void printMethod() {
        // Example code here
    }
}
```
```java
    System.out.println("Printed in Superclass.");
}
public class Subclass extends Superclass {
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass");
    }
}
public static void main(String[] args) {
    Subclass s = new Subclass();
    s.printMethod();
}
```

7.1.3 Generics

“Generics are a powerful feature of the Java programming language.

– They improve the type safety of your code, making more of your bugs detectable at compile time.” [http://docs.oracle.com/javase/tutorial/java/index.html]

“In a nutshell, generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

– Much like the more familiar formal parameters used in method declarations, Type parameters provide a way for you to re-use the same code with different inputs.

– The difference is that the inputs to formal parameters are values, While the inputs to type parameters are types”

7.2 Collection (without using generics)

• Collection Interface
  – A collection represents a group of objects, known as its elements
• List Interface
  – An ordered collection. The user can access elements by their integer index (position in the list), and search for elements in the list.
• ArrayList
  – Resizable-array implementation of the List interface
  – Implements the size, isEmpty, get, set, iterator, and listIterator methods
- **LinkedList**
  - Doubly-linked list implementation of the List interface
  - Implements the size, isEmpty, get, set, iterator, and listIterator methods

```
// interface java.util.Collection
< interface >
java.util.Collection

// interface java.util.List
< interface >
java.util.List

interface List extends Collection {...}
```

- **List versus List<T>**

  Nice:
  ```java
  List list = new ArrayList();
  list.add("hello");
  String s = (String) list.get(0);
  ```

  Nicer (because no typecasts, and better type-checking):
  ```java
  List<String> list = new ArrayList<String>();
  list.add("hello");
  String s = list.get(0); // no cast
  ```

  What will happen if you put different classes of objects in a List, then invoke findMax() on the objects in that list?
  - You would get a runtime error if the isLargerThan() method for one class doesn’t accept objects in the other class as a parameter.
7.3 Visibility Rules, from the Java Tutorial

<table>
<thead>
<tr>
<th>Access Levels</th>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

- “The first data column indicates whether the class itself has access to the member defined by the access level.
- “The second column indicates whether [other] classes in the same package as the class (regardless of their parentage) have access to the member.
- “The third column indicates whether subclasses of the class declared outside this package have access to the member.
- “The fourth column indicates whether all classes have access to the member.”

7.4 Tips on Choosing an Access Level

“If other programmers use your class, you want to ensure that errors from misuse cannot happen.
- Access levels can help you do this.
  “Use the most restrictive access level that makes sense for a particular member.
  “Use private unless you have a good reason not to.
  “Avoid public fields except for constants.
  - (Many of the examples in the tutorial use public fields. This may help to illustrate some points concisely, but is not recommended for production code.)
  - Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.”

7.5 Packages

“Definition: A package is a grouping of related types providing access protection and name space management. “Note that types refer to classes, interfaces, enumerations, and annotation types.”
Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.”

Purpose: “To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.”

7.5.1 Creating a Package

“To create a package, you

– Choose a name for the package (naming conventions are discussed in the next section) and

– Put a package statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

“The package statement (for example, package graphics;) must be the first line in the source file.

– There can be only one package statement in each source file, and it applies to all types in the file.”

7.5.2 One public type per file!

“If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file.

– For example, you can
– Define public class Circle in the file Circle.java,
– Define public interface Draggable in the file Draggable.java,
– Define public enum Day in the file Day.java, and so forth.

“You can include non-public types in the same file as a public type (this is strongly discouraged, unless the non-public types are small and closely related to the public type),

– But only the public type will be accessible from outside of the package.

– All the top-level, non-public types will be package private.”

This rule makes it easy for the class loader, and the human programmer, to find the definition for a public type.

– The name of a package determines the directory in which the files of this package should be
The name of a public type determines the name of the file in which the type’s definition must be found.

### 7.5.3 The default package

“If you do not use a package statement, your type ends up in an unnamed package.

- Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process.
- Otherwise, classes and interfaces belong in named packages.”

### 7.5.4 Package Naming Conventions

“With programmers worldwide writing classes and interfaces using the Java programming language,

- It is likely that many programmers will use the same name for different types.
- [Because the fully-qualified name of each class includes the package name, e.g. java.awt.Rectangle,
- There is no naming conflict] unless two independent programmers use the same name for their packages.
- What prevents this problem [of name conflict]? Convention

**Naming Conventions**

Package names are written in all lower case to avoid conflict with the names of classes or interfaces. Companies use their reversed Internet domain name to begin their package names

- for example, com.example.mypackage for a package named mypackage created by a programmer at example.com.

Name collisions that occur within a single company need to be handled by convention within that company. Packages in the Java language itself begin with java. or javax.

### 7.5.5 External references

“To use a public package member from outside its package, you must do one of the following:

- Refer to the member by its fully qualified name,
- Import the package member, [or]
– Import the member’s entire package.”
– The fully qualified name for class C in package p is p.C
– To import class C from package p, you write import p.C
– To import an entire package p, you write import p.*

Java gives you a great deal of control over the “name space” of your program:
– You control the import of external names (by your import statements)
– You control the export of your names (by restricting visibility, in packages and in
inheritances).

7.5.6 Keywords for Control of Inheritance

Every subclass will
– inherit all superclass members that are declared as public or protected
By contrast,
– private members are not inherited (but may be accessible through super.)
– The default visibility is “package-private” – inherited by subclasses within the same
package, but not inherited by subclasses that are declared outside the package.

No subclass can
– Override static methods, or
– Override final methods.

Any subclass may
– Add new members (= fields or methods), or
– Override any non-static, non-final method in the superclass.

We say a method is overridden in a subclass, if any of its superclasses has a method of the same
signature (= name, plus the number and types of parameters) and return type.
– Note that overriding does not absolutely prevent access. A reference to the superclass
member is still possible (e.g. with super) if this member is visible.
Lecture No. 8
Java Implementation: Part 3

8.1 Keywords for Control of Inheritance

Every subclass will
- Inherit all superclass members that are declared as public or protected

By contrast,
- Private members are not inherited (but may be accessible through super.)
- The default visibility is “package-private” – inherited by subclasses within the same package, but not inherited by subclasses that are declared outside the package.

No subclass can
- Override static methods, or
- Override final methods.

Any subclass may
- Add new members (= fields or methods), or
- Override any non-static, non-final method in the superclass.

We say a method is overridden in a subclass, if any of its super classes has a method of the same signature (= name, plus the number and types of parameters) and return type.

Note that overriding does not absolutely prevent access. A reference to the superclass member is still possible (e.g. with super) if this member is visible.

8.2 Statically or Dynamically Typed

Programming languages generally offer some sort of type system, and can be described as being either statically typed or dynamically typed

With a statically typed language, compile-time checks are carried out to determine whether variable usage is valid.

In a dynamically typed language, variables are not associated with a type and are simply names that can be assigned arbitrary values.
8.2.1 Java - a statically typed language

Every variable name is bound

– To a static type (at compile time, by means of a data declaration), and
– Either to a dynamic type or null, depending on its current value

The type restricts the values that can be bound to this variable.

```java
int x = 2.3;
```

The type also restricts the messages that can be sent using the variable.

```java
int x = 2; (Vector) x.add(0x37);
```

Restrictions are checked at compile-time.

– The compiler will not issue code if it detects a violation.
– Java is a “type-safe” language: its compile-time checking restricts the amount of damage that can be done by careless or malicious programmers.

8.2.2 Static Typing Restrictions

A reference variable of static type T can refer to an instance of class T or to an instance of any of T’s subclasses. A type is a restriction on the values that can be taken by a variable, and a subclass is a stricter restriction – so there can be no type error when a value in a subtype of T is assigned to a variable of type T.

Through a reference variable of static type T, the set of messages that can be sent using that variable are the methods defined by class T and its super classes.

– This typing rule allows inherited methods to be accessed via T, in contexts where the names of these methods are visible.
– There might be many subclasses of T, each defining different methods with the same name – so T can’t be used to refer to any of these subclass methods.

Recall: a variable’s static type is fixed at compile time, but its dynamic type may vary at run-time.
Example: Static Binding of Instance Variables

```java
class Base {
    public int x = 10;
}

public class Derived extends Base {
    public int y = 20;
}
```

```java
//Case 1:
Base b1 = new Base();
System.out.println("b1.x= " + b1.x);

//Case 2:
Derived b2 = new Derived();
System.out.println("b2.x= " + b2.x);
System.out.println("b2.y= " + b2.y);

//Case 3:
Base b3 = new Derived();
// System.out.println("b3.x= " + b3.x);
```

8.2.3 Static Binding – Hiding a Field

“Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different.

“Within the subclass, the field in the superclass cannot be referenced by its simple name. Instead, the field must be accessed through super, which is covered in the next section.

“Generally speaking, we don't recommend hiding fields as it makes code difficult to read.” [The Java Tutorials]
8.3 Review: Fields & Variables

The Java Tutorials makes a careful distinction between fields and variables.

– Not many programmers use these terms carefully.
– You won’t understand the Java Tutorials, in full technical detail, unless you understand its definitions!

In the Variables page of the Language Basics Lesson:

– “Instance Variables (Non-Static Fields) Technically speaking, objects store their individual states in ‘non-static fields’, … also known as instance variables …
– “Class Variables (Static Fields) A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.
– “Local Variables Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. … There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared — which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.
– “Parameters … The important thing to remember is that parameters are always classified as ‘variables’ not ‘fields’. … [In addition to methods, ] other parameter-accepting constructs … [include] constructors and exception handlers …”

8.4 Dynamic Binding

If a method is overridden, then the compiler may not be able to resolve a reference to that method. The runtime search for an overridden method begins with the dynamic type.

– If this type doesn’t implement the method (i.e. it neither introduces nor overrides the method), then the search progresses up the hierarchy, until the method is found.
– Static type-checking ensures that an implementation will be found (unless the class was changed, and re-compiled, after the type-check.)
If a method is overridden, then the compiler may not be able to resolve a reference to that method. The runtime search for an overridden method begins with the dynamic type.

- If this type doesn’t implement the method (i.e. it neither introduces nor overrides the method), then the search progresses up the hierarchy, until the method is found.

- Static type-checking will ensure that an implementation will be found -- unless the class was changed, and re-compiled, after the type-check!

8.5 Conversions of Primitive Types

Widening conversions

Wider assignment, e.g. int i = 2; float x = i;

Wider casting, e.g. int i = 2; double d = (double) i;

Explicitly casting can make your code more readable

Narrowing conversions

Narrow assignment

a compile-time error!
float f = 2.0;
int i = f;

- Narrow casting
  - a loss of information!

float f = 2.0;
int i = (int) f;

**Object Type Conversions**

Widening conversions
- Wider object reference assignment conversion (allowed)
- Wider object reference casting (optional: improves readability)

**Object Types**

Narrowing conversions
- Narrow object reference assignment – Compile-time error!
- Narrow object reference casting – no compilation error, but…

  The cast may throw an error at run-time, to avoid assigning an out-of-range value!

### 8.6 Overriding, hiding, and overloading methods

“An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.”

“If a subclass defines a class method with the same signature as a class method in the superclass, the method in the subclass hides the one in the superclass.

- “The distinction between hiding and overriding has important implications.
- The version of the overridden method that gets invoked is the one in the subclass.
- The version of the hidden method that gets invoked depends on whether it is invoked from the superclass or the subclass.”
“Overloaded methods are differentiated by the number and the type of the arguments passed into the method.”

– “The compiler does not consider return type when differentiating methods, so you cannot declare two methods [in the same class] with the same signature even if they have a different return type.

– “Note: Overloaded methods should be used sparingly, as they can make code much less readable.”

Lecture No. 9
Java Implementation: Part 4

9.1 Enum Types

“An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include:

– Compass directions (values of NORTH, SOUTH, EAST, and WEST) and
– The days of the week

“Because they are constants, the names of an enum type's fields are in uppercase letters.

“… define an enum type by using the enum keyword.

– For example, you would specify a days-of-the-week enum type as:

```java
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```
“You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and datasets where you know all possible values at compile time—for example,

- The choices on a menu,
- Command line flags, and so on.”

### 9.2 Memory Allocation

We use a reference variable to refer to instantiated objects. The value in a reference variable is, essentially, a pointer to an object.

- A special value (null) indicates that there is no object.
- The runtime system (the JVM) interprets reference values as an index into a heap—an area of memory that is set aside, by the JVM, for storing instantiated objects.
- Formally: the range of allowable values for a reference variable is defined by its reference type. This is a static property.

The reference type of o1 is Object. This means it can point to any instance of Object, or to any instance of any subclass of Object.

The new operator allocates sufficient memory on the heap to store all the fields of the object it is instantiating.
A model of Java’s type system (for reference)


Variables, revisited

“The Java programming language defines the following kinds of variables: …”

[Variables page of the Language Basics Lesson]
Object Identity

If two reference variables have the same value, they are pointing to the same object.

- This relationship is called “object identity”.
- You can test it with the == operator.

**9.3 Equality test: object identity**

A box that contains 7 items is not identical to any other box that contains 7 items.

- But… we would say “3 + 4 equals 7”.

If we want to know whether two boxes are equivalent (= have the same value), we might have to open up the boxes and look inside.

- The equals() method is implemented as == in Object.
- You should override equals(), if you define a subclass in which the “natural definition” for equality differs from the equals() it inherits.
9.4 The `hashCode()` Method

“The Object class, in the java.lang package, sits at the top of the class hierarchy tree.

– Every class is a descendant, direct or indirect, of the Object class.
– Every class you use or write inherits the instance methods of Object.
– You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class.

“The value returned by `hashCode()` is the object's hash code, which is the object's memory address in hexadecimal.

“By definition, if two objects are equal, their hash code must also be equal.
– If you override the equals() method, you change the way two objects are equated and Object's implementation of `hashCode()` is no longer valid.
– Therefore, if you override the equals() method, you must also override the `hashCode()` method as well.”

The `hashCode()` method returns an int.

– Hashcodes are used in HashSet, HashMap, and some other Collection classes which use a hashing algorithm.
– These classes will give incorrect results, if equal instances in a Collection have different hashcodes.
– They will have poor performance, if many unequal instances share the same hashcode.

**String Equality – be careful…**

Strings are immutable.

– None of the String methods will modify the value of an existing instance; instead, a new String instance is created, and returned.

Some strings are “interned” (= accessible by a hash lookup, at runtime).

– You may get a reference to an existing String instance when you ask for a new String.
Then again, you might not...

```
String s1 = "Apple";
String s2 = "Apple";
System.out.println("s1==s2:" + (s1==s2));
System.out.println("s1.equals(s2):" + s1.equals(s2));
```

```
String s3 = new String("Apple");
String s4 = new String("Apple");
System.out.println("s3==s4:" + (s3==s4));
System.out.println("s3.equals(s4):" + s3.equals(s4));
```

Moral: you should use `equals()`, and not `==`, to test Strings for equality.

### 9.5 Other Overridable Object Methods

Object has two other methods you might want to override

- `toString()`: returns a String representation of the object
- `clone()`: create a copy of an existing object

```java
class Object {
...

    public boolean equals(Object obj) {
        return (this == obj);
    }

    public String toString() {
        return getClass().getName() ...;
    }

    protected Object clone() throws CloneNotSupportedException {
        ...;
    }
}
```
The getClass() method

You cannot override getClass().

- Can you see why this isn’t allowed?

```java
public class Object {
    ...
    // Returns the runtime class of an object
    public final Class getClass() {
        ...
    }
    ...
}
Point p1 = new Point(10, 20);
Class c = p1.getClass();
System.out.println(c.getName());
System.out.println(c.getSuperclass().getName());
java.lang.Object
```

9.6 Cloning

The clone() method in the Object class

- Throws an exception, if the class of this object does not implement the interface Cloneable
- Creates an object of the same type as the original object
- Initialises the clone's instance variables to the same values as the original object's instance variables

This is a shallow copy: any objects that are referenced by instance variables will not be cloned.

If an object references another object, then you might want to override clone() so that
- It always throws an exception (i.e. is uncloneable), or
- It clones the other object, and references it from the clone of the original -- so that the clone of the original can be modified or destroyed without affecting the original.

9.7 Nested Classes

Definition: A class defined inside another class.
Motivation: Some classes only make sense in the context of another enclosing class. Examples:
– An Enumeration or Iterator object cannot exist by itself. It makes sense only in association with a collection being enumerated/iterated
– A GUI event handler cannot exist by itself. It makes sense only in association with the GUI component for which it handles events.

Reference: the Writing an Event Listener Lesson of the Java Tutorials.

– Nested classes define, and enforce, a composition relationship between the outer class and its inner classes:

```
public class MyRegularClass {
  ...
  class MyInnerClass {
    ...
  }
}
```

9.7.1 Nested Classes: Some Details

“A nested class is a member of its enclosing class.
– Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private.
– Static nested classes do not have access to other members of the enclosing class.

“As a member of [its outer class], a nested class can be declared private, public, protected, or package private.
– (Recall that outer classes can only be declared public or package private.)”

“There are two additional types of inner classes.
– You can declare an inner class within the body of a method.
Such a class is known as a local inner class.
– You can also declare an inner class within the body of a method without naming it.
These classes are known as anonymous inner classes.
You will encounter such classes in advanced Java programming.”

Lecture No. 10
Frameworks

10.1 Framework
Generic software platform for a certain type of applications, consists of parts that are found in many apps of that type:

- Libraries with APIs (classes with methods etc.)
- Ready-made extensible programs ("engines")
- Sometimes also tools (e.g. for development, configuration, content)

Often evolved by developing many apps of that type and reusing code more and more

Characteristics:
- Reusable: the parts can be used for many apps of that type
- Extensible: developers can add their own app-specific code
- Inversion of Control: framework often calls your code

Framework Examples
10.2 Extensibility

All frameworks can be extended to cater for app-specific functionality. A framework is intended to be extended to meet the needs of a particular application.

Common ways to extend a framework:

- Extension is carried out by sub-classing, overriding methods, and implementing interfaces
- Plug-ins: framework can load certain extra code in a specific format
- Within the framework language:
  - Subclassing & overriding methods
  - Implementing interfaces
  - Registering event handlers

10.3 Inversion of Control

A framework employs an inverted flow of control between itself and its clients. When using a framework, one usually just implements a few callback functions or specializes a few classes, and then invokes a single method or procedure. The framework does the rest of the work for you, invoking any necessary client callbacks or methods at the appropriate time and place.

- i.e. "Don't call us, we'll call you."
- "Leave the driving to us."

Example: Java's Swing and AWT classes.

- They have a huge amount of code to manage the user interface, and there is inversion of control because you start the GUI framework and then wait for it to call your listeners.
Inversion of Control

- **Traditional Program Execution**
  - Application calls an I/O library routine
  - Application calls a sort routine

- **Inversion of Control**
  - Framework generates a window resize event
  - Framework generates a paint event
  - Framework generates a button click event

The app has control over the execution flow, calling library code when it needs to.

The framework has control over the execution flow, calling app code for app-specific behavior.

Example: Java Applets
10.4 What is Java Frameworks?

Frameworks are large bodies (usually many classes) of prewritten code to which you add your own code to solve a problem in a specific domain. In Java technology there are so many frameworks that help the programmers to build complex applications easily. Examples:

- GUI Framework: eg Java's Swing and AWT classes
- Collection Framework/library
  - It is a unified architecture for representing and manipulating collections
  - It contains Interfaces, Implementations and algorithms

Collections Framework

- The Java Collections Framework provides the following benefits:
  - Reduces programming effort
  - Concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.
  - Increases program speed and quality
  - It provides high-performance, high-quality implementations of useful data structures and algorithms
  - Fosters software reuse
  - New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Frameworks VS libraries

- Framework uses your code because it is usually the framework that is in control.
- It means the framework controls the sequence and logic of some operation and calls your code to provide certain details
- You make use of a framework by calling its methods, inheritance, and supplying "callbacks", listeners, etc
- Note: although sometimes large libraries are referred to as frameworks, this is probably not the most common use of the term.
Advantages

– Reuse can save cost and time
– Higher level of abstraction
  ▶ Frameworks provides a standard working system through which user can develop the desired module of application or complete application instead of developing lower level details.
– Developers can devote more time in developing the software requirement
– Reduced maintenance cost (if the framework is maintained by someone else)

Disadvantages

– Can lead to code bloat
  ▶ Framework may contain lots of unused code
  ▶ May need to use several frameworks
– Cost of learning a framework
  ▶ Spend more time in assessing the concept, function and its uses in developing the program.
– Licensing cost (for commercial frameworks)
– A generic ‘one-size-fits-all’ does not work so efficiently for any specific software. There is need to extend framework with specific code to develop any specific software.

10.5 GUI Programming Concepts

Conventional programming:
– Sequence of operations is determined by the program
– What you want to happen, happens when you want it

Event-driven programming:
– Sequence of operations is determined by the user’s interaction with the application’s interface
– Anything that can happen, happens at any time

GUI Design Concepts

– A wealth of information creates a poverty of attention
  - Herbert Simon
Principles of good GUI Design
- IBM's Design concepts
- Saul Greenberg's HCI pages
- Tim's HCI notes

GUI Programming Concepts in Java
- Java GUI has components
- Windows GUI has controls
- Unix GUI has widgets
- Examples: labels, buttons, check boxes, radio buttons, text input boxes, pull down lists
- Swing components: JLabel, JButton, JCheckBox, JRadioButton, JTextField, JTextArea, JComboBox

Java GUI history: the AWT
- AWT(JDK 1.0, 1.1):
  - Abstract Window Toolkit
- Package: java.awt, java.awt.event
- Heavyweight components using native gui system elements
- Used for applets until most browsers supported jre 1.2

Swing in Java
- Swing(Java 2, JDK 1.2+)
  - Lightweight components that do not rely on the native GUI or OS
  - “look and feel” of Swing components
  - Are identical on different platforms
  - Can be customized
  - Swing inherits from AWT
  - AWT still used for events, layouts

Swing Components in Java
- Advanced GUI support. E.g. Drag-and-drop
- Package names: javax.swing, javax.swing.event
- Components inherit from jcomponent
- Components are added to a top-level container: JFrame, JDialog, or JApplet.
Lecture No. 11
GUI Programming in Java

11.1 GUI Programming Concepts

Conventional programming:
– Sequence of operations is determined by the program
– What you want to happen, happens when you want it

Event-driven programming:
– Sequence of operations is determined by the user’s interaction with the application’s interface
– Anything that can happen, happens at any time

GUI Design Concepts
– A wealth of information creates a poverty of attention
  - Herbert Simon
– Principles of good GUI Design
– IBM’s Design concepts
– Saul Greenberg’s HCI pages
– Tim’s HCI notes

GUI Programming Concepts in Java
– Java GUI has components
– Windows GUI has controls
– Unix GUI has widgets
– examples: labels, buttons, check boxes, radio buttons, text input boxes, pull down lists
– Swing components: JLabel, JButton, JCheckBox, JRadioButton, JTextField, JTextArea, JComboBox

Java GUI history: the AWT
– AWT(JDK 1.0, 1.1):
  Abstract Window Toolkit
– Package: java.awt, java.awt.event
– Heavyweight components using native gui system elements
– Used for applets until most browsers supported JRE 1.2

**Swing in Java**
– Swing(Java 2, JDK 1.2+)
– Lightweight components that do not rely on the native GUI or OS
– “look and feel” of Swing components
– Are identical on different platforms
– Can be customized
– Swing inherits from AWT
– AWT still used for events, layouts

**Swing Components in Java**
– Advanced GUI support. E.g. Drag-and-drop
– Package names: javax.swing, javax.swing.event
– Components inherit from JComponent
– Components are added to a top-level container: JFrame, JDialog, or JApplet.

**Running a Swing application**
– Java -Dswing.aatext=true MySwingClass
– The option sets the system property "swing.aatext" to "true" to enable anti-aliasing for every JComponent
– javaw runs a GUI without the console window

**11.2 Basic GUI Programming Steps in Java**
– Declare a container and components
– Add components to one or more containers using a layout manager
– Register event listener(s) with the components
– Create event listener method(s)
– Example: JFrameDemo.java, JFrameDemoTM.java
– Container: a screen window/applet window/panel that groups and arranges GUI components
– GUI component: an object with visual representation
– Swing containers: JFrame, JApplet, JPanel
– AWT containers: Frame, Applet, Panel
11.3 GUI Programming: The Java Approach

Event-driven programming
- A piece of code (i.e. Event handler) is attached to a gui component
- An event handler is called when an event (e.g. A mouse click) is activated / fired

The Delegation Event Model in Java
- Processing of an event is delegated to an object (the listener) in the program

Event-driven Programming in Java
- Event source: a GUI component that generates / fires an event
- Event: a user interaction (e.g. A click on the button)
- Event listener: an object that has encapsulated event handlers to react to an event

11.4 Event Handling in Java

The delegation event model
- A GUI element “delegates” the processing of an event to another piece of code (i.e. An event handler)
- The event source generates/fires an event and “sends” it to event listeners

The delegation event model
- Event listeners must be registered with an event source in order to receive notification
- Example: JButtonDemo.java, JButtonDemo2.java

Registration of an event listener
- Write a class that implements an <event name>Listener interface
- Create an instance of that class (i.e. an event listener)
- Register the listener with a GUI component:
  
  add<event name>Listener ( <an event listener> )

A listener interface has a list of standard event handlers (i.e. methods)

API documentation
- Java.awt.event
- Event classes
- Adapter classes
- Listener interfaces
Different ways of coding the event listeners
- Use of another top-level class: jButtonDemo
- Anonymous inner classes:
  - JButtonDemo2, JFrameDemo, JFrameDemo2
- Use of an adapter class: JFrameDemo3

JFrameDemoTM2.java uses named inner classes and shows how to consolidate window closing from two different events.

11.5 Handling of Mouse Events

MouseListener
- mousePressed( ), mouseClicked( )
- mouseReleased( ), mouseEntered( )
- mouseExited( )

MouseMotionListener
- mouseDragged( ), mouseMoved( )
- MouseAdapter, MouseMotionAdapter

Example: MouseDemo.java

Lecture No. 12

User Interface Design

12.1 User interface design

User interface design is the process of designing effective interfaces for software systems. The main objectives of user interface design are:

- To suggest some general design principles for user interface design
- To explain different interaction styles
- To introduce styles of information presentation
- To describe the user support which should be built-in to user interfaces
- To introduce usability attributes and system approaches to system evaluation
The user interface

- System users often judge a system by its interface rather than its functionality.
- A poorly designed interface can cause a user to make catastrophic errors.
- Poor user interface design is the reason why so many software systems are never used.

Graphical user interfaces

- Most users of business systems interact with these systems through graphical interfaces, although, in some cases, legacy text-based interfaces are still used.

GUI characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Multiple windows allow different information to be displayed simultaneously on the user’s screen.</td>
</tr>
<tr>
<td>Icons</td>
<td>Icons different types of information. On some systems, icons represent files; on others, icons represent processes.</td>
</tr>
<tr>
<td>Menus</td>
<td>Commands are selected from a menu rather than typed in a command language.</td>
</tr>
<tr>
<td>Pointing</td>
<td>A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interest in a window.</td>
</tr>
<tr>
<td>Graphics</td>
<td>Graphical elements can be mixed with text on the same display.</td>
</tr>
</tbody>
</table>

12.2 GUI Advantages

- They are easy to learn and use.
- Users without experience can learn to use the system quickly.
- The user may switch quickly from one task to another and can interact with several different applications.
- Information remains visible in its own window when attention is switched.
- Fast, full-screen interaction is possible with immediate access to anywhere on the screen.
12.3 User interface design process

- Analyse and understand user activities
- Produce paper-based design prototype
- Evaluate design with end-users
- Design prototype
- Produce dynamic design prototype
- Evaluate design with end-users
- Executable prototype
- Implement final user interface

12.4 User Interface design principles

- UI design must take account of the needs, experience and capabilities of the system users
- Designers should be aware of people’s physical and mental limitations (e.g. limited short-term memory) and should recognise that people make mistakes
- UI design principles underlie interface designs although not all principles are applicable to all designs

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User familiarity</td>
<td>The interface should use terms and concepts which are drawn from the experience of the people who will make most use of the system.</td>
</tr>
<tr>
<td>Consistency</td>
<td>The interface should be consistent in that, wherever possible, comparable operations should be activated in the same way.</td>
</tr>
<tr>
<td>Minimal surprise</td>
<td>Users should never be surprised by the behaviour of a system.</td>
</tr>
<tr>
<td>Recoverability</td>
<td>The interface should include mechanisms to allow users to recover from errors.</td>
</tr>
<tr>
<td>User guidance</td>
<td>The interface should provide meaningful feedback when errors occur and provide context-sensitive user help facilities.</td>
</tr>
<tr>
<td>User diversity</td>
<td>The interface should provide appropriate interaction facilities for different types of system user.</td>
</tr>
</tbody>
</table>

Figure 12.1: User interface design principles
Design principles

User familiarity
– The interface should be based on user-oriented terms and concepts rather than computer concepts. For example, an office system should use concepts such as letters, documents, folders etc. rather than directories, file identifiers, etc.

Consistency
– The system should display an appropriate level of consistency. Commands and menus should have the same format; command punctuation should be similar, etc.

Minimal surprise
– If a command operates in a known way, the user should be able to predict the operation of comparable commands

Recoverability
– The system should provide some resilience to user errors and allow the user to recover from errors. This might include an undo facility, confirmation of destructive actions, ‘soft’ deletes, etc.

User guidance
– Some user guidance such as help systems, on-line manuals, etc. should be supplied

User diversity
– Interaction facilities for different types of user should be supported. For example, some users have seeing difficulties and so larger text should be available

Lecture No. 13

User Interface Design 2

13.1 User-System Interaction

Two problems must be addressed in interactive systems design
– How should information from the user be provided to the computer system?
– How should information from the computer system be presented to the user?
User interaction and information presentation may be integrated through a coherent framework such as a user interface metaphor.

**Interaction styles**
- Direct manipulation
- Menu selection
- Form fill-in
- Command language
- Natural language

**Advantages and disadvantages**

<table>
<thead>
<tr>
<th>Interaction style</th>
<th>Main advantages</th>
<th>Main disadvantages</th>
<th>Application examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct manipulation</td>
<td>Fast and intuitive interaction</td>
<td>May be hard to implement</td>
<td>Video games, CAD systems</td>
</tr>
<tr>
<td></td>
<td>Easy to learn</td>
<td>Only suitable where there is a visual metaphor for tasks and objects</td>
<td></td>
</tr>
<tr>
<td>Menu selection</td>
<td>Avoids user error</td>
<td>Slow for experienced users</td>
<td>Most general-purpose systems</td>
</tr>
<tr>
<td></td>
<td>Little typing required</td>
<td>Can become complex if many menu options</td>
<td></td>
</tr>
<tr>
<td>Form fill-in</td>
<td>Simple data entry</td>
<td>Takes up a lot of screen space</td>
<td>Stock control, Personal loan</td>
</tr>
<tr>
<td></td>
<td>Easy to learn</td>
<td></td>
<td>processing</td>
</tr>
<tr>
<td>Command language</td>
<td>Powerful and flexible</td>
<td>Hard to learn</td>
<td>Operating systems, Library</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Poor error management</td>
<td>information retrieval systems</td>
</tr>
<tr>
<td>Natural language</td>
<td>Accessible to casual users</td>
<td>Requires more typing</td>
<td>Timetable systems, WWW information</td>
</tr>
<tr>
<td></td>
<td>Easily extended</td>
<td>Natural language understanding systems are unreliable</td>
<td>retrieval systems</td>
</tr>
</tbody>
</table>

**Direct manipulation advantages**
- Users feel in control of the computer and are less likely to be intimidated by it
- User learning time is relatively short
- Users get immediate feedback on their actions so mistakes can be quickly detected and corrected

**Direct manipulation problems**
- The derivation of an appropriate information space model can be very difficult
Given that users have a large information space, what facilities for navigating around that space should be provided?

Direct manipulation interfaces can be complex to program and make heavy demands on the computer system.

### 13.2 Control Panel Interface

**1- Menu systems**
- Users make a selection from a list of possibilities presented to them by the system.
- The selection may be made by pointing and clicking with a mouse, using cursor keys or by typing the name of the selection.
- May make use of simple-to-use terminals such as touchscreens.

**Advantages of menu systems**
- Users need not remember command names as they are always presented with a list of valid commands.
- Typing effort is minimal.
- User errors are trapped by the interface.
- Context-dependent help can be provided. The user’s context is indicated by the current menu selection.

**Problems with menu systems**
- Actions which involve logical conjunction (and) or disjunction (or) are awkward to represent.
Menu systems are best suited to presenting a small number of choices. If there are many choices, some menu structuring facility must be used.

Experienced users find menus slower than command language.

**2- Form-based interface**

Command interfaces

- User types commands to give instructions to the system e.g. UNIX
- May be implemented using cheap terminals.
- Easy to process using compiler techniques
- Commands of arbitrary complexity can be created by command combination
- Concise interfaces requiring minimal typing can be created

**Problems with command interfaces**

- Users have to learn and remember a command language. Command interfaces are therefore unsuitable for occasional users
- Users make errors in command. An error detection and recovery system is required
- System interaction is through a keyboard so typing ability is required
3. Command languages
– Often preferred by experienced users because they allow for faster interaction with the system
– Not suitable for casual or inexperienced users
– May be provided as an alternative to menu commands (keyboard shortcuts), in some cases, a command language interface and a menu-based interface are supported at the same time

4. Natural language interfaces
– The user types a command in a natural language. Generally, the vocabulary is limited and these systems are confined to specific application domains (e.g. timetable enquiries)
– NL processing technology is now good enough to make these interfaces effective for casual users but experienced users find that they require too much typing

5. Multiple user interfaces

13.3 Information presentation
Information presentation is concerned with presenting system information to system users. The information may be presented directly (e.g. text in a word processor) or may be transformed in some way for presentation (e.g. in some graphical form). The Model-View-Controller approach is a way of supporting multiple presentations of data.
Model-view-controller

Information presentation

Static information
- Initialised at the beginning of a session, it does not change during the session
- May be either numeric or textual

Dynamic information
- Changes during a session and the changes must be communicated to the system user
- May be either numeric or textual

13.4 Information Display Factors
- Is the user interested in precise information or data relationships?
- How quickly do information values change? Must the change be indicated immediately?
- Must the user take some action in response to a change?
Is there a direct manipulation interface?

Is the information textual or numeric? Are relative values important?

**Alternative information presentations**

**Analogue vs. Digital presentation**

**Digital presentation**

- Compact - takes up little screen space
- Precise values can be communicated

**Analogue presentation**

- Easier to get an 'at a glance' impression of a value
- Possible to show relative values
- Easier to see exceptional data values
Dynamic information display

- Dial with needle
- Pie chart
- Thermometer
- Horizontal bar

Displaying relative values

<table>
<thead>
<tr>
<th>Pressure</th>
<th>Temperature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 100 200 300 400</td>
<td>0 25 50 75 100</td>
</tr>
</tbody>
</table>

Textual highlighting

The filename you have chosen has been used. Please choose another name.

Ch. 16 User interface design

OK Cancel
13.5 Data Visualization

Data visualization is concerned with techniques for displaying large amounts of information. Visualisation can reveal relationships between entities and trends in the data. Possible data visualisations are:

- Weather information collected from a number of sources
- The state of a telephone network as a linked set of nodes
- Chemical plant visualised by showing pressures and temperatures in a linked set of tanks and pipes
- A model of a molecule displayed in 3 dimensions
- Web pages displayed as a hyperbolic tree

**Colour displays**

- Colour adds an extra dimension to an interface and can help the user understand complex information structures
- Can be used to highlight exceptional events
- Common mistakes in the use of colour in interface design include:
  - The use of colour to communicate meaning
  - Over-use of colour in the display

**Colour use guidelines**

- Don't use too many colours
- Use colour coding to support use tasks
- Allow users to control colour coding
- Design for monochrome then add colour
- Use colour coding consistently
- Avoid colour pairings which clash
- Use colour change to show status change
- Be aware that colour displays are usually lower resolution
**User support**

- User guidance covers all system facilities to support users including on-line help, error messages, manuals etc.
- The user guidance system should be integrated with the user interface to help users when they need information about the system or when they make some kind of error
- The help and message system should, if possible, be integrated

**Help and message system**

**Error messages**

- Error message design is critically important. Poor error messages can mean that a user rejects rather than accepts a system
- Messages should be polite, concise, consistent and constructive
- The background and experience of users should be the determining factor in message design
Lecture No. 14

Usability and User Interface Design

14.1 What is Usability?

Usability is the:

- Ease of learning
- Ease of use
- Ease of remembering
- Subjective satisfaction
- Efficiency of use
- Effectiveness of use

14.2 What is Usability Engineering?

Usability Engineering is:

- Processes to build “Usability” into products
- Various methods can be used throughout the design lifecycle
– Methods can be incorporated into design process easily
– Methods maintain focus on user throughout design

**Why Isn’t UE Done Regularly?**

Developers believe
– It takes too long
– Is too expensive
– Is not critical to development
– They can afford to learn about user problems during “live” use

Also developers may not know how to do it

**Benefits of UE to an Organization**

– Reduce training costs
– Reduce development costs
– Identify and fix problems early
– Reduce support costs; minimize need for
– Support personnel/help desks
– Fixes, maintenance, upgrades
– Enhance organization’s reputation - positive “word-of-mouth” trade
– Larger numbers of “hit” and “return visit” rates

**Benefits of UE to the User**

– Less time to complete work
– Greater success with tasks
– Increased user satisfaction

**14.3 Typical Web-site Usability Problems**

**Navigation**

– Knowing where you are
– Finding what you want

**Structure of web site**

**Layout**
– Needs sufficient white space
– Use of large graphics

“The Google problem”

14.4 User Centered Design

UCD is a dialog between the customer and the designer

Rules of thumb:
– Get to know and understand the users.
– Build an application, applying usability principles.
– Test designs by observing users in a real work setting (environment and work load).

14.5 Usability Principles

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout</td>
<td>The interface should be a series of areas on the screen that are used consistently for different purposes—for example, a top area for commands and navigation, a middle area for information to be input or output, and a bottom area for status information.</td>
</tr>
<tr>
<td>Content awareness</td>
<td>Users should always be aware of where they are in the system and what information is being displayed.</td>
</tr>
<tr>
<td>Aesthetics</td>
<td>Interfaces should be functional and inviting to users through careful use of white space, colors, and fonts. There is often a tradeoff between including enough white space to make the interface look pleasing without losing so much space that important information does not fit on the screen.</td>
</tr>
<tr>
<td>User experience</td>
<td>Although ease of use and ease of learning often lead to similar design decisions, there is sometimes a tradeoff between the two. Novice users or infrequent users of software will prefer ease of learning, whereas frequent users will prefer ease of use.</td>
</tr>
<tr>
<td>Consistency</td>
<td>Consistency in interface design enables users to predict what will happen before they perform a function. It is one of the most important elements in ease of learning, ease of use, and aesthetics.</td>
</tr>
<tr>
<td>Minimal user effort</td>
<td>The interface should be simple to use. Most designers plan on having no more than three mouse clicks from the starting menu until users perform work.</td>
</tr>
</tbody>
</table>

14.6 GUI Design is Multi-Disciplinary

A team includes
– Analyst
– Designer
Basic Principles

Assume users:

- Have not read the manual
- Have not attended training
- Do not have external help readily at hand
- So… All controls should be clear and understandable and placed in an intuitive location on the screen.

Usability Design Process

Use Scenario Development

- You already know about use cases
- Use scenarios are just a single thread through a use case – much simpler!
UI Structure/Flow of Control

15.1 What is Usability?

- Ease of learning
- Ease of use
- Ease of remembering
- Subjective satisfaction
- Efficiency of use

FIGURE 12-7  An Example Window Navigation Diagram

Lecture No. 15

USABILITY and

USER INTERFACE DESIGN (CONTD…)

Software Construction  |  VCIIT
Usability Design Process

15.2 Use Scenario Development

You already know about use cases. Use scenarios are just a single thread through a use case – much simpler!

UI Structure/Flow of Control

− This matters much less than it used to. Why?
− What can you do about it?
Users often jump right in to the middle of a web-site (via search). This means even more opportunities to be “lost in space”. What to do? Always make it clear: what the web-site is for, who you are, how to go to the home page.

15.3 Interface Standards Design

- The interface standards are the basic design elements that are common across the screens, forms, and reports within the system.

Interface Design Prototyping

- Storyboard
- HTML Prototype
- Language Prototype
- Interface Evaluation
- Heuristic
- Walkthrough
- Interactive
- Formal Usability Testing

Creating a UI Prototype

Low-fidelity prototypes are good! e.g.

- Paper prototype
- Visio/HTML/PPT/… prototype

Example:
Heuristic Evaluation

Set of guidelines for creating usable GUIs

Interface/Dialogue Design

The team needs to worry about:

- Layout (of buttons, text, table data, …)
- Structuring data entry (tab order)
- Controlling data input (validation and format controls)
- Feedback (prompting, status, warning, and error messages)
- Dialogue sequencing

What Is a Dialogue?

A sequence of interactions between the system and a user

Dialogue design involves:

- Designing a dialogue sequence
- Building a prototype
- Assessing usability

GUI Design Guidelines

- User in control
- Consistency
- Personalization & Customization
- Forgiveness
- Feedback
- Aesthetics & Usability

User in Control

- Rather the user’s perception of control
- No mothering principle – user should feel that they initiate actions
– Feedback is necessary for this to work

**Consistency**

The conformance to the GUI vendor’s standards

– A GUI developer must not be too creative and innovative in the interface design.

The conformance to the naming, coding and other GUI-related standards developed internally by the organization

– This includes the naming and coding of menus, action buttons, screen fields, etc.

– It also includes standards for the placement of screen objects and consistent use of other GUI elements across applications

**Personalization & Customization**

GUI personalization is customization for personal use

– e.g. a user reorders and resizes columns in a grid display and saves these changes as preferences

GUI customization is an administrative task of tailoring the software to different groups of users

– e.g. when the program operates differently for novice and advanced users

**Forgiveness**

A good interface allows users to experiment and make mistakes in a “forgiving” way

Forgiveness encourages interface exploration because the user is allowed to take erroneous routes that can be “rolled back” (even to the starting point if necessary).

– Forgiveness implies facilities such as a multi-level undo operation and the ability to cancel long-running actions

**Feedback**

– The feedback guideline is a spin-off of the first guideline – the user in control.

– To be “in control” involves knowing what’s going on when the control is temporarily with the program.
– The developer should build visual and/or audio cues for every user event.
– Hourglass, wait indicator…

**Aesthetics & Usability**

– Aesthetics is about visual appeal.
– Usability is about the ease, simplicity, efficiency, reliability in using the interface.

The issues to consider include

– The fixation and movement of the human eye,
– The use of colors,
– The sense of balance and symmetry,
– The alignment and spacing of elements,
– The sense of proportion,
– The grouping of related elements, etc.
– The responsiveness of the system

Simplicity – additional and related guideline

**Symmetry: Law of Proximity**

How many groups are there in this image?

How do you know?
What do the radio buttons apply to?

How do you know?

What do you see in the image to the right?

What is it really?
Color Example

Left is color-safe, middle is as seen by green-insensitive dichromat, right is as seen by a red-insensitive dichromat.

15.4 Ten Golden Usability Rules

1. Use simple and natural dialog/language.

   User interfaces should be simplified as much as possible, since every additional feature or item of information on a screen is one more thing to learn, one more thing to misunderstand, and one more thing to search through. And the GUI should not overdo it with colour (= Less is more).

2. Use language that fits the user group.

   The language should be based on user’s language and not on the system-oriented terms. Translation from one language to another is more than just words. Time, currency, phrases, metaphors, measurements, etc. must fit the culture of the user group.

3. Minimise the load on short-term memory.

   Users should not have to remember information from one part of the dialogue to another. Instructions to the use of the system should be visible or easily retrievable whenever appropriate.
4. **Make the graphical user interface coherent and consistent.**

   The same action should always have the same effect. Users will feel confident in using the system, and they will be encouraged to try out exploratory learning because they already have part of the knowledge needed to operate new parts of the system.

5. **Give the ability to use shortcuts.**

   Accelerators – unseen by the novices – may often speed up the interaction for the expert’s users, such as: function keys, command keys, macros, etc.

6. **Give feedback to the user's actions.**

   The system should always keep users informed about what is going on, through appropriate feedback within reasonable time i.e.

   – Seconds user feels the system is responding immediately;
   – S. Is about the limit user’s flow of thought will stay interrupted;
   – 10 seconds is about the limit to keep user’s attention focused on the dialogue; so if the user has to wait longer the system should give feedback about the waiting time. Warning messages should be used when the user is going to perform an irreversible action.

7. **Avoid error situations.**

   Ask the user if s/he really wants to do the action especially if the action leads to serious consequences. Avoid having too similar commands in the interface.

   Commands whose actions are opposite should not be placed close to each other.

8. **Give clear exit marks.**

   Users often choose actions by mistake and will need a clearly marked ”emergency exit” to leave the unwanted state without having to go through an extended dialogue. Cancel should be offered when the operation will take a long time. In the navigation bars, the users should be able to back up.
9. **Give clear and understandable error messages.**

Simple rules for error messages:

- Should be phrased clearly and avoid obscure codes
- Messages should be precise, not vague or general
- Messages should constructively help the user to solve the problem
- Messages should be polite and not intimidate the user or put the blame explicitly on the user
- “Illegal, fatal action, job aborted!” Phrases should not be used.

10. **Give clear help and understandable documentation.**

- Information should be easy to search; it should be focused on the user’s task. List the concrete steps that should be carried out. The help manual should not be too large.

**Lecture 16**

**Software Design Patterns**

**16.1 Background**

- Search for recurring successful designs – emergent designs from practice (via trial and error)
- Supporting higher levels of reuse (i.e., reuse of designs) is quite challenging
- Described in Gama, Helm, Johnson, Vlissides 1995 (i.e., “gang of 4 book”)
- Based on work by Christopher Alexander (an Architect) on building homes, buildings and towns.
- Design patterns represent solutions to problems that arise when developing software within a particular context. E.g., problem/solution pairs within a given context
- Describes recurring design structures
- Describes the context of usage
- Patterns capture the static and dynamic structure and collaboration among key participants in software designs
– Especially good for describing how and why to resolve nonfunctional issues
– Patterns facilitate reuse of successful software architectures and designs.

### 16.2 Origins of Design Patterns

“Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice”

#### Elements of Design Patterns

Design patterns have four essential elements:

– Pattern name
– Problem
– Solution
– Consequences

#### Pattern Name

A handle used to describe:

– A design problem
– Its solutions
– Its consequences
– Increases design vocabulary
– Makes it possible to design at a higher level of abstraction
– Enhances communication

– “The Hardest part of programming is coming up with good variable [function, and type] names.”

#### Problem

– Describes when to apply the pattern
– Explains the problem and its context
– May describe specific design problems and/or object structures
– May contain a list of preconditions that must be met before it makes sense to apply the pattern
Solution
Describes the elements that make up the

- Design
- Relationships
- Responsibilities
- Collaborations
- Does not describe specific concrete implementation
- Abstract description of design problems and how the pattern solves it

Consequences
Results and trade-offs of applying the pattern

Critical for:

- Evaluating design alternatives
- Understanding costs
- Understanding benefits of applying the pattern

Includes the impacts of a pattern on a system’s:

- Flexibility
- Extensibility
- Portability

Design Patterns are NOT

- Designs that can be encoded in classes and reused as is (i.e., linked lists, hash tables)
- Complex domain-specific designs (for an entire application or subsystem)

They are:

- “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context”

Where they are used

Object-Oriented programming languages [and paradigm] are more amenable to implementing design patterns.

Procedural languages: need to define

- Inheritance
- Polymorphism
16.3 Describing Design Patterns

- Graphical notation is generally not sufficient
- In order to reuse design decisions the alternatives and trade-offs that led to the decisions are critical knowledge
- Concrete examples are also important
- The history of the why, when, and how set the stage for the context of usage

Design Patterns

Describe a recurring design structure
- Defines a common vocabulary
- Abstracts from concrete designs
- Identifies classes, collaborations, and responsibilities
- Describes applicability, trade-offs, and consequences

Example: Stock Quote Service
Observer Pattern

Intent:

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Key forces:

- There may be many observers
- Each observer may react differently to the same notification
- The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject

Structure of Observer Pattern
Collaborations in Observer Pattern

Design Pattern Descriptions

- Name and Classification: Essence of pattern
- Intent: What it does, its rationale, its context
- AKA: Other well-known names
- Motivation: Scenario illustrates a design problem
- Applicability: Situations where pattern can be applied
- Structure: Class and interaction diagrams
- Participants: Objects/classes and their responsibilities
- Collaborations: How participants collaborate
- Consequences: Trade-offs and results
- Implementation: Pitfalls, hints, techniques, etc.
- Sample Code
- Known Uses: Examples of pattern in real systems
- Related Patterns: Closely related patterns
16.4 Types of Patterns

Creational patterns:
– Deal with initializing and configuring classes and objects

Structural patterns:
– Deal with decoupling interface and implementation of classes and objects
– Composition of classes or objects

Behavioral patterns:
– Deal with dynamic interactions among societies of classes and objects
– How they distribute responsibility

1- Creational Patterns

Abstract Factory:
– Factory for building related objects

Builder:
– Factory for building complex objects incrementally

Factory Method:
– Method in a derived class creates associates

Prototype:
– Factory for cloning new instances from a prototype

Singleton:
– Factory for a singular (sole) instance

2- Structural Patterns

Adapter:
– Translator adapts a server interface for a client

Bridge:
– Abstraction for binding one of many implementations

Composite:
– Structure for building recursive aggregations

Decorator:
– Decorator extends an object transparently
Facade:
  – Simplifies the interface for a subsystem

Flyweight:
  – Many fine-grained objects shared efficiently.

Proxy:
  – One object approximates another

3- Behavioral Patterns

Chain of Responsibility:
  – Request delegated to the responsible service provider

Command:
  – Request is first-class object

Iterator:
  – Aggregate elements are accessed sequentially

Interpreter:
  – Language interpreter for a small grammar

Mediator:
  – Coordinates interactions between its associates

Memento:
  – Snapshot captures and restores object states privately

Observer:
  – Dependents update automatically when subject changes

State:
  – Object whose behavior depends on its state

Strategy:
  – Abstraction for selecting one of many algorithms

Template Method:
  – Algorithm with some steps supplied by a derived class

Visitor:
  – Operations applied to elements of a heterogeneous object structure
## 16.5 Design Pattern Space

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Factory method</td>
<td>Creational</td>
</tr>
<tr>
<td></td>
<td>Adapter (class)</td>
<td>Structural</td>
</tr>
<tr>
<td></td>
<td>Interpreter</td>
<td>Behavioral</td>
</tr>
<tr>
<td></td>
<td>Template method</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Abstract factory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Adapter (object)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Bridge</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Composite</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Decorator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Façade</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flyweight</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Proxy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chain of responsibility</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Command</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Iterator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mediator</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Memento</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Observer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Strategy</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Visitor</td>
<td></td>
</tr>
</tbody>
</table>

### Categorization Terms
Scope is the domain over which a pattern applies
- Class Scope: relationships between base classes and their subclasses (static semantics)
- Object Scope: relationships between peer objects

Some patterns apply to both scopes.

#### Class::Creational
- Abstracts how objects are instantiated
- Hides specifics of the creation process
- May want to delay specifying a class name explicitly when instantiating an object
- Just want a specific protocol

#### Example
Use **Factory Method** to instantiate members in base classes with objects created by subclasses
- Abstract Application class: create application-specific documents conforming to a particular Document type
- Application instantiates these Document objects by calling the factory method
DoMakeDocument
- Method is overridden in classes derived from Application
- Subclass DrawApplication overrides DoMakeDocument to return a DrawDocument object

Class:: Structural
Use inheritance to compose protocols or code

Example:
**Adapter Pattern:** makes one interface (Adaptee’s) conform to another
- Gives a uniform abstraction of different interfaces
- Class Adapter inherits privately from an Adaptee class
- Adapter then expresses its interface in terms of the Adaptee’s.

Class:: Behavioral
Captures how classes cooperate with their subclasses to satisfy semantics.

Example:
**Template Method:** defines algorithms step by step.
- Each step can invoke an abstract method (that must be defined by the subclass) or a base method.
- Subclass must implement specific behavior to provide required services

Object Scope
- Object Patterns all apply various forms of non-recursive object composition.
- Object Composition: most powerful form of reuse
- Reuse of a collection of objects is better achieved through variations of their composition, rather than through sub classing.

Object:: Creational
Abstracts how sets of objects are created

Example:
**Abstract Factory:** create “product” objects through generic interface
- Subclasses may manufacture specialized versions or compositions of objects as allowed by this generic interface

User Interface Toolkit: 2 types of scroll bars (Motif and Open Look)
- Don’t want to hard-code specific one; an environment variable decides
Class Kit:
- Encapsulates scroll bar creation (and other UI entities);
- An abstract factory that abstracts the specific type of scroll bar to instantiate
- Subclasses of Kit refine operations in the protocol to return specialized types of scroll bars.
- Subclasses MotifKit and OpenLookKit each have scroll bar operation.

Object:: Structural
Describe ways to assemble objects to realize new functionality
- Added flexibility inherent in object composition due to ability to change composition at run-time
- Not possible with static class composition

Example:
 Proxy: acts as convenient surrogate or placeholder for another object.
- Remote Proxy: local representative for object in a different address space
- Virtual Proxy: represent large object that should be loaded on demand
- Protected Proxy: protect access to the original object

Object:: Behavioral
Describes how a group of peer objects cooperate to perform a task that can be carried out by itself

Example:
Strategy Pattern: objectifies an algorithm
Text Composition Object: support different line breaking algorithms
- Don’t want to hard-code all algs into text composition class/subclasses
Objectify different algs and provide them as Compositor subclasses (contains criteria for line breaking strategies)
Interface for Compositors defined by Abstract Compositor Class
- Derived classes provide different layout strategies (simple line breaks, left/right justification, etc.)
Instances of Compositor subclasses couple with text composition at run-time to provide text layout
Whenever text composition has to find line breaks, forwards the responsibility to its current...
Compositor object.

16.6 When to Use Patterns

Solutions to problems that recur with variations
– No need for reuse if problem only arises in one context

Solutions that require several steps:
– Not all problems need all steps
– Patterns can be overkill if solution is a simple linear set of instructions

Solutions where the solver is more interested in the existence of the solution than its complete derivation
– Patterns leave out too much to be useful to someone who really wants to understand
– They can be a temporary bridge

What makes it a Pattern?

A Pattern must:
– Solve a problem and be useful
– Have a context and can describe where the solution can be used
– Recur in relevant situations
– Provide sufficient understanding to tailor the solution
– Have a name and be referenced consistently

Benefits of Design Patterns
– Design patterns enable large-scale reuse of software architectures and also help document systems
– Patterns explicitly capture expert knowledge and design trade-offs and make it more widely available
– Patterns help improve developer communication
– Pattern names form a common vocabulary
– Patterns help ease the transition to OO technology

Drawbacks to Design Patterns
– Patterns do not lead to direct code reuse
– Patterns are deceptively simple
– Teams may suffer from pattern overload
Patterns are validated by experience and discussion rather than by automated testing.
Integrating patterns into a software development process is a human-intensive activity.

Suggestions for Effective Use

Do not recast everything as a pattern.

- Instead, develop strategic domain patterns and reuse existing tactical patterns
  - Institutionalize rewards for developing patterns
  - Directly involve pattern authors with application developers and domain experts
  - Clearly document when patterns apply and do not apply
  - Manage expectations carefully.

Lecture No. 17

Threads

17.1 Threads

Single and Multithreaded Processes

![Diagram showing single-threaded and multithreaded processes](image)
Benefits of Threads
- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

User Threads
Thread management done by user-level threads library
Examples
- POSIX Pthreads
- Mach C-threads
- Solaris threads

Kernel Threads
Supported by the Kernel
Examples
- Windows
- Solaris
- Tru64 UNIX
- BeOS
- Linux

Scheduling
17.2 Multithreading Models

- Many-to-One
- One-to-One

**Many-to-One**

Many user-level threads mapped to single kernel thread. Used on systems that do not support kernel threads.

Examples:

- Solaris Green Threads
- GNU Portable Threads

**One-to-One**

Each user-level thread maps to kernel thread.

Examples

- Windows
- Linux
17.3 Threading Issues
Semantics of fork() and exec() system calls
Does fork() duplicate only the calling thread or all threads?

Thread cancellation
Terminating a thread before it has finished
Two general approaches:

- Asynchronous cancellation terminates the target thread immediately
- Deferred cancellation allows the target thread to periodically check if it should be cancelled

Signal handling
Signals are used in UNIX systems to notify a process that a particular event has occurred

A signal handler is used to process signals
- Signal is generated by particular event
- Signal is delivered to a process
- Signal is handled

Options:
- Deliver the signal to the thread to which the signal applies
- Deliver the signal to every thread in the process
- Deliver the signal to certain threads in the process
– Assign a specific thread to receive all signals for the process

**Thread pools**

Create a number of threads in a pool where they await work

Advantages:
– Usually slightly faster to service a request with an existing thread than create a new thread
– Allows the number of threads in the application(s) to be bound to the size of the pool

17.4 Various Implementations

**PThreads**
– A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
– API specifies behavior of the thread library, implementation is up to development of the library
– Common in UNIX operating systems (Solaris, Linux, Mac OS X)

**Windows Threads**

Implements the one-to-one mapping

Each thread contains
– A thread id
– Register set
– Separate user and kernel stacks
– Private data storage area

The register set, stacks, and private storage area are known as the context of the threads

**Linux Threads**
– Linux refers to them as *tasks* rather than *threads*
– Thread creation is done through clone() system call
– clone() allows a child task to share the address space of the parent task (process)

**Java Threads**

Java threads may be created by:
– Extending Thread class
– Implementing the Runnable interface

Java threads are managed by the JVM.
Traditional View of a Process

Process = process context + code, data, and stack

Alternate View of a Process

Process = thread + code, data, and kernel context
A **Process with Multiple Threads**

Multiple threads can be associated with a process

- Each thread has its *own* logical control flow (sequence of PC values)
- Each thread *shares* the same code, data, and kernel context
- Each thread has its own thread id (TID)

**Logical View of Threads**

Threads associated with a process form pool of peers

- Unlike processes, which form tree hierarchy

**17.5 Threads vs. Processes**

How threads and processes are similar
- Each has its own logical control flow
- Each can run concurrently
- Each is context switched

How threads and processes are different
- Threads share code and data, processes (typically) do not
- Threads are somewhat cheaper than processes
- Process control (creating and reaping) is twice as expensive as thread control
- Linux/Pentium III numbers:
  - ~20K cycles to create and reap a process
  - ~10K cycles to create and reap a thread

Posix Threads (Pthreads) Interface

Pthreads: Standard interface for ~60 functions that manipulate threads from C programs

Creating and reaping threads
- pthread_create, pthread_join

Determining your thread ID
- pthread_self

Terminating threads
- pthread_cancel, pthread_exit
- exit [terminates all threads], return [terminates current thread]

Synchronizing access to shared variables
- pthread_mutex_init, pthread_mutex_[un]lock
- pthread_cond_init, pthread_cond_[timed]wait
The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *howdy(void *vargp);

int main() {
    pthread_t tid;
    Pthread_create(&tid, NULL, howdy, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *howdy(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```

Execution of Threaded “hello, world”

17.6 Pros and Cons of Thread-Based Designs

Threads take advantage of multicore/multi-CPU H/W
Easy to share data structures between threads
– E.g., logging information, file cache

Threads are more efficient than processes

Unintentional sharing can introduce subtle and hard-to-reproduce errors!

– Ease of data sharing is greatest strength of threads

– Also greatest weakness!

**Shared Variables in Threaded C Programs**

Question: Which variables in a threaded C program are shared variables?

– Answer not as simple as “global variables are shared” and “stack variables are private”

Requires answers to the following questions:

– What is the memory model for threads?

– How are variables mapped to memory instances?

– How many threads reference each of these instances?

Lecture No. 18

**Software Testing**

18.1 **White-Box Testing**

“This strategy derives test data from an examination of the program’s logic

– “(and often, unfortunately, at the neglect of the specification)”

What is the overall strategy or “gold standard” for white-box testing?

– “Causing every statement in the program to execute at least once”?

– No… this is “highly inadequate”.

18.2 **Test phases**

**Acceptance Testing** – this checks if the overall system is functioning as required.

**Unit testing** – this is basically testing of a single function, procedure, class.

**Integration testing** – this checks that units tested in isolation work properly when put together.

**System testing** – here the emphasis is to ensure that the whole system can cope with **real data**, monitor system performance, test the system’s error handling and recovery routines.

**Regression Testing** – this checks that the system preserves its functionality after maintenance
Testing Tools

Unit Testing

Unit Tests are tests written by the developers to test functionality as they write it. Each unit test typically tests only a single class, or a small cluster of classes. *Unit tests are typically written using a unit testing framework, such as JUnit (automatic unit tests).*

Target errors not found by Unit testing:

- Requirements are mis-interpreted by developer.
- Modules don’t integrate with each other

**Unit testing: a white-box approach**

Testing based on the coverage of the executed program (source) code.

Different coverage criteria:

- statement coverage
- path coverage
- condition coverage
- definition-use coverage
- ...

It is often the case that it is not possible to cover all code. For instance:

- for the presence of dead code (not executable code)
- for the presence of not feasible path in the CFG
Acceptance Testing

Acceptance Tests are specified by the customer and analyst to test that the overall system is functioning as required (Do developers build the right system?).

- Acceptance tests typically test the entire system, or some large chunk of it.
- When all the acceptance tests pass for a given user story (or use case, or textual requirement), that story is considered complete.
- At the very least, an acceptance test could consist of a script of user interface actions and expected results that a human can run.
- Ideally acceptance tests should be automated, either using the unit testing framework (Junit), or a separate acceptance testing framework (Fitnesse).

Used to judge if the product is acceptable to the customer

- Coarse grained tests of business operations
- Scenario/Story-based (contain expectations)
- Simple:
  - Happy paths (confirmatory)
  - Sad paths
  - Alternative paths (deviance)

Acceptance testing: a black-box approach

1. describe the system using a Use-Cases Diagram
   * a use-case of that diagram represents a functionality implemented by the system
2. detail each use-case with a textual description of, e.g., its pre-post conditions and flow of events
   * events are related to: (i) the interactions between system and user; and (ii) the expected actions of the system
   * a flow of events is composed of basic and alternate flows
3. define all instances of each use-case (scenarios) executing the system for realizing the functionality
4. define, at least, one test case for each scenario
5. (opt) define additional test cases to test the interaction between use-cases.
How to select input values?
Different approaches can be used:

Random values:
- For each input parameter we randomly select the values

Tester Experience:
- For each input we use our experience to select relevant values to test

Domain knowledge:
- We use requirements information or domain knowledge information to identify relevant values for inputs

Combinatorial Testing:
Test all possible combination of the inputs is often impossible

- e.g., method(a:int,b:int,c:int) .. how many combinations?
- with 10 values per input: $10^3=1000$
- with 100 values per input: $100^3=1000000$

Selection of relevant combinations is important

- Pairwise testing (aka 2-way): cover all combinations for each pair of inputs

- $<a,b> <a,c> <b,c> = 10^2 + 10^2+10^2=300$
- don’t care about the value of the third input

18.3 Iterative Software Development
19.1 White-Box Testing

“This strategy derives test data from an examination of the program’s logic

– “(and often, unfortunately, at the neglect of the specification)”

What is the overall strategy or “gold standard” for white-box testing?

– “Causing every statement in the program to execute at least once”?

– No… this is “highly inadequate”.

19.2 A Gold Standard for White-Box Testing

Exhaustive Path Testing: a gold standard for White-Box Testing

– “If you execute, via test cases, all possible paths of control flow through the program,
then possibly the program has been completely tested.”
Testing all possible paths of control flow is not enough to prove correctness in a white-box test, so how can this be a gold standard?

– Testing all possible inputs is not enough to prove correctness in a black-box test, yet this is still the gold standard for black-box testing.

– The justification is the other way around! If you don’t exercise all control paths, then a test case for this path may reveal an obvious bug in this path. Someone might ask: why didn’t you test that path? How would you respond?

Testing all paths is often infeasible.

– The gold-standard for black-box testing (of testing all possible inputs) is almost always infeasible.

– Even if you know you can’t possibly “reach the gold”, you can still use this standard to measure your progress!

**19.3 What is a “Possible Path of Control Flow”?**

The concept of a “unique logic path”, where a “logic path” is the sequence of instructions executed when a program is given a particular input.

– If the program has a conditional branch, then it has multiple logic paths (assuming that an input can be found which causes the program to branch, and another input which causes the program not to branch).

– Your goal, as a white-box tester, is to devise an input which will “force” the program to take each important path.

– We define “path” informally in this course, with reference to a flowchart as on the following slide.

**19.4 White-Box Testing: An Example**

Devise a test set for this code. (Inputs: a, b; Outputs: a, x, y.)
\[ x := a + b; \]
\[ y := a \times b; \]
\[ \text{while } (y > a) \{ \]
\[ \quad a := a + 1; \]
\[ \quad x := a + b \]
\[ \}

\[ a = 1, b = 1 \text{? } \text{Yes, this is “red” input… output: } a = 1, x = 2, y = 1. \]

\[ x := a + b; \]
\[ y := a \times b; \]
\[ \text{while } (y > a) \{ \]
\[ \quad a := a + 1; \]
\[ \quad x := a + b \]
\[ \}

\[ a = 1, b = 2 \text{? } \text{Hmmm… } x = 3, y = 2; \text{ then } a = 2, x = 4. \]

\[ x := a + b; \]
\[ y := a \times b; \]
\[ \text{while } (y > a) \{ \]
\[ \quad a := a + 1; \]
\[ \quad x := a + b \]
\[ \}

\[ a = 1, b = 3 \text{? } \text{Hmmm… } x = 4, y = 3; a = 2, x = 7; a = 3, x = 10. \]
19.5 Cyclomatic Complexity

It provides a quantitative measure of the logical complexity of a program. It defines the number of independent paths in the basis set. Provides an upper bound for the number of tests that must be conducted to ensure all statements have been executed at least once.

Can be computed three ways

- The number of regions
\[ V(G) = E - N + 2, \text{ where } E \text{ is the number of edges and } N \text{ is the number of nodes in graph } G \]

\[ V(G) = P + 1, \text{ where } P \text{ is the number of predicate nodes in the flow graph } G \]

Results in the following equations for the example flow graph:

- Number of regions = 4
- \( V(G) = 14 \text{ edges} - 12 \text{ nodes} + 2 = 4 \)
- \( V(G) = 3 \text{ predicate nodes} + 1 = 4 \)

**A Second Flow Graph Example**

```c
int function1(void)
{
    int x = 0;
    int y = 19;
    A: x++;
    if (x > 999)
        goto D;
    if (x % 11 == 0)
        goto B;
    else goto A;
    B: if (x % y == 0)
        goto C;
    else goto A;
    C: printf("%d\n", x);
    goto A;
    D: printf("End of list\n");
    return 0;
}
```

**A Sample Function to Diagram and Analyze**

```c
int function2(int y)
{
    int x = 0;
    while (x <= (y * y))
    {
        if ((x % 11 == 0) &&
            (x % y == 0))
        {
            printf("%d", x);
            x++;
        } // End if
        else if ((x % y == 0) ||
            (x % y == 2))
        {
            printf("%d", y);
            x = x + 2;
        } // End else
        else
            printf("\n");
    } // End while
    printf("End of list\n");
    return 0;
}
```
When should we stop?

Don’t add a test unless it has a reasonable chance of exposing a new bug.

- A loop that “works correctly” on two iterations is not very likely to show an obvious problem on the third iteration, so we might stop after testing the green path.
- A problem: we don’t know what this program is supposed to do, so how can we choose a “correct” set of output values for each of our test cases in this example?!
- This example is an extreme form of white-box testing, in which there is no specification, aside from the code we’re looking at.
- In Regression Testing, we gain confidence that the current version of a system has the same behaviour as prior versions.
- Regression testing is very useful when you’re porting a program to a new system.
- Does “it work the same way” on both systems?
- Regression testing is also useful when confirming that recent changes to a program haven’t re-introduced a bug.
- (Hmmm… should I add a new test case every time I identify a new bug?)
- Don’t add a test unless it has a reasonable chance of exposing a new bug.
- A loop that “works correctly” on two iterations is not very likely to show an obvious problem on the third iteration, so we might stop after testing the green path.
- A problem: we don’t know what this program is supposed to do, so how can we choose a “correct” set of output values for each of our test cases in this example?!
- This example is an extreme form of white-box testing, in which there is no specification, aside from the code we’re looking at.
- In Regression Testing, we gain confidence that the current version of a system has the same behaviour as prior versions.
- Regression testing is very useful when you’re porting a program to a new system.
- Does “it work the same way” on both systems?
- Regression testing is also useful when confirming that recent changes to a program haven’t re-introduced a bug.
- (Hmmm… should I add a new test case every time I identify a new bug?)
If you test a path...

Can you conclude that there are no bugs on a path you have tested? Of course not! … but let’s list some reasons why we may miss some bugs, then think about whether we can write test cases to cover these…

– The output we specify for our test case may be incorrect
– because we were doing a regression test against a buggy version of our program,
– because we interpreted the specification carelessly,
– or because the specification wasn’t clear.
– What can we do to decrease the number of incorrect test cases we write?
– There may be data-sensitive calculations in the path, and we’re only testing a single point of what may be a very non-linear function.
– If the path contains “x = a+b” but it should contain “x = a*b”, our case won’t reveal the error if our test input has a=0 or b=0.
– Should we write more test cases for paths with data-sensitive computations?
– The computation on this path may be non-deterministic.
– If a program is multi-threaded, then the value computed on one path may depend greatly on what path another thread is following, and on how far along that path the other thread has already moved. (Do you understand multi-threading?)

Lecture 20
Software Quality

20.1 Extreme Testing (XT)

XT is the testing methodology of an XP development process
– The philosophy is “extreme”.
– Maximise the main defect-finding activities, stop other testing activities.
Main activities: unit testing and acceptance testing.
Extreme unit testing (XUT)
– Do only the main activities in a unit test.
Do them at maximum levels, within the time & resource constraints.

(What are these main activities?)

Extreme acceptance testing (XAT)

- Do only the main activities in an acceptance test
- Do them at maximum levels.

**Extreme Unit Testing**

XUT has two “simple rules”:

- All code modules must have unit tests before coding begins
- All code modules must pass unit tests before being released into production.

Nothing new here, except the insistence on writing unit tests before coding.

- Wow, that’s a disciplined approach! Would you do this willingly, or would you be tempted to “code early, on the sly” (when you think your manager isn’t looking)?
- Note: if you’re writing executable tests, then I’d say you’re coding.
- You can write assertions in Java or Junit.
- You might be programming in a language with goal-directed evaluation (e.g. Icon):
  specify the outcome (in restricted settings) and let the computer figure it out!
- In a futuristic/AI development scenario, you could be “programming by example” i.e. goal-directed programming without any sharply-defined restrictions.

**Benefits of “Test-First Coding”**

- You gain confidence that your code will meet its specification.
- You express the end result of your code before you start coding.
- You better understand the application’s specification and requirements.
- You may initially implement simple designs and confidently refactor the code later to improve performance [*and elegance – important for maintainability*] without worrying about breaking the specification.

**The “Shining Point” of XP**

- “The practice of creating unit tests first is the shining point of the XP methodology, as it forces you to understand the specification to resolve ambiguities *before* you begin coding.”
- ?Really? What keeps me from writing unit tests hastily, without resolving ambiguities?
“If you create a robust set of black-box tests, this demonstrates that you understand the specification. Only after demonstrating this, should you start to code.” (I’m in the “quality school” of testing, as defined in a question on Quiz 2 this weekend)

So… there must be some quality-control on the unit tests.

What’s the main activity in this quality control?

### 20.2 The Importance of Automated Testing

“Manually running unit tests, even for the smallest application, can be a daunting task.

As the application grows, you may generate hundreds or thousands of unit tests.

Therefore you typically used an automated testing suite to ease the burden of constantly running unit tests.”

Main functions of an automated testing suite:

- Script the tests, then run all or part of them
- Create reports and classify the bugs: may be useful in future development

The “testing code base” becomes as valuable as the software application itself, so it should be

- Stored in a code repository,
- With adequate backups & security

### 20.3 Extreme acceptance testing (XAT)

Purpose of XAT: to determine, with a minimum of effort and time, whether the application is acceptable to the customer.

Is XP unsuitable for use whenever there is more than one customer?

Some software development methodologies (as taught in information-systems departments) acknowledge that

Stakeholders are often deeply conflicted about requirements on new IT systems, and

IT system specifications are “levers of change” in an organisation, implying that

Acceptance-testing decisions by an employee may be “over-rulled” by management.

Let’s assume the XAT team has a fully-empowered and well-informed “customer” at their disposal! Then…

“… Customers, not you or your programming partners, conduct the acceptance tests.”
“In this manner, customers provide the unbiased verification that the application meets their needs.”

**Relation of XAT to user stories**

- “Customers create the acceptance tests from user stories.”
- “The ratio of user stories to acceptance tests is usually one to many.
- “That is, more than one acceptance test may be needed for each user story.”
- ? This sounds like quite a burden on the customer! I wonder how many XP projects are actually doing this?

**Automation of XAT**

- “Acceptance tests in XT may or may not be automated.
- “For example, an unautomated test is required when the customer must validate that a user-input screen meets its specification with respect to color and screen layout.”
- Hmmm… a layout & colour test could be fully automated, with some image-processing techniques, if the specification is very precise. However a spec that’s a one-line “user story” is will require some subjective measurement of attributes such as “easy to use”, “legible”, “attractive”.
- “An example of an automated test is when the application must calculate some payroll values using data input via some data source such as a flat file to simulate production values.”
- Hmm, this sounds like the acceptance test in 3C.

**20.4 XAT: a Validation or a Verification**

In XAT, the customer is asked whether or not the system produces valid output.

- System validation: commonly defined as “Are we building the right thing for you?”
- System verification: “Are we building it right?”, that is, does the system meet its specifications?
- Requirements validation: “Do the requirements specify a system that you, the customer, would want to use – assuming we can build it?”
- Requirements verification: “Do the requirements make sense?”, that is, could our dev team understand them well enough to implement them, or are they too vague, contradictory, or
infeasible?

Because customers aren’t allowed to change their stories during an acceptance test, I’d say XAT is system verification.

– Any new stories should be prioritised into the release schedule.
– My question: If there are no new stories during XAT, is the current system valid?

20.5 The Peril of Changing Requirements

If previously-accepted requirements are changed, the development may make little or no “forward progress”

– Developers must revise unit tests, and then recode the units, to conform to the new requirements

If previously-accepted requirements are “cast in concrete”, then the project may fail as soon as the “mistake” is discovered.

– In my (very limited) experience, end-users rarely know what they want until they have fiddled with a prototype. Then the stories change rapidly!

20.6 Software updates are hazardous

– When software is in the field, it is very hazardous to change any of its features.
– This is true even when a feature is reported as a "bug" by some stakeholders, and the QA team agrees that it is a bug.
– Other stakeholders may have become accustomed to the buggy behaviour, and are likely to be confused, annoyed, or even angered when it is "fixed".
  – In a sports analogy, this hazard is called
  – "Changing the rules after the game has started".
– My advice: unless there's a major security risk, a major legal risk, or a major dissatisfaction among stakeholders with a software product,
  – User-visible behaviour ("look and feel") should remain constant.
  – Even during a major version-step, feature-change should be minimised.
– I say this because I believe most users don’t want to learn new features, adjust their behaviour, or modify their expectations… but … if a change is “really cool” then users will happily “invest” significant time and money in order to gain a novel experience!
Lecture No. 21
Software Change, Evolution and Maintenance

21.1 Software Evolution
It is impossible to produce system of any size which does not need to be changed. Once software is put into use, new requirements emerge and existing requirements changes as the business running that software changes.
Parts of the software may have to be modified to correct errors that are found in operation, improve its performance or other non-functional characteristics. All of this means that, after delivery, software systems always evolve in response to demand for change.

21.2 Software change
Software change is inevitable
- New requirements emerge when the software is used
- The business environment changes
- Errors must be repaired
- New equipment must be accommodated
- The performance or reliability may have to be improved
- A key problem for organisations is implementing and managing change to their legacy systems

21.3 Software change strategies

Software maintenance
- Changes are made in response to changed requirements but the fundamental software structure is stable

Architectural transformation
- The architecture of the system is modified generally from a centralised architecture to a distributed architecture

Software re-engineering
No new functionality is added to the system but it is restructured and reorganised to facilitate future changes.

These strategies may be applied separately or together.

### 21.4 Program evolution dynamics

Program evolution dynamics is the study of the processes of system change. After major empirical study, Lehman and Belady proposed that there were a number of ‘laws’ which applied to all systems as they evolved.

There are sensible observations rather than laws. They are applicable to large systems developed by large organisations. Perhaps less applicable in other cases.

#### Lehman’s laws

<table>
<thead>
<tr>
<th>Law</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuing change</td>
<td>A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.</td>
</tr>
<tr>
<td>Increasing complexity</td>
<td>As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.</td>
</tr>
<tr>
<td>Large program evolution</td>
<td>Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors are approximately invariant for each system release.</td>
</tr>
<tr>
<td>Organisational stability</td>
<td>Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.</td>
</tr>
<tr>
<td>Conservation of familiarity</td>
<td>Over the lifetime of a system, the incremental change in each release is approximately constant.</td>
</tr>
</tbody>
</table>

#### Applicability of Lehman’s laws

This has not yet been established.

They are generally applicable to large, tailored systems developed by large organisations.

It is not clear how they should be modified for

- Shrink-wrapped software products
- Systems that incorporate a significant number of COTS components
- Small organisations
Medium sized systems

21.5 Types of Software Maintenance

Software Maintenance

– Software maintenance is the general process of changing a system after it has been diverted.
– The change may be simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancement to correct specification error or accommodate new requirements.

Maintenance Characteristics

– We need to look at maintenance from three different viewpoints: [PRE2004]
– the activities required to accomplish the maintenance phase and the impact of a software engineering approach (or lack thereof) on the usefulness of such activities
– the costs associated with the maintenance phase
– the problems that are frequently encountered when software maintenance is undertaken

Types of Maintenance

Maintenance to repair software faults
– Changing a system to correct deficiencies in the way meets its requirements

Maintenance to adapt software to a different operating environment
– Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation

Maintenance to add to or modify the system’s functionality
– Modifying the system to satisfy new requirements
Maintenance Effort Distribution

- Fault repair (17%)
- Software adaptation (18%)
- Functionality addition or modification (65%)

Development vs. Maintenance

<table>
<thead>
<tr>
<th>Not directly linked to the real world</th>
<th>Directly driven by the real world</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freedom</td>
<td>Constrained by existing system</td>
</tr>
<tr>
<td>Defects have no immediate effect</td>
<td>Defects disrupt production</td>
</tr>
<tr>
<td>Methods available</td>
<td>System not using current methods</td>
</tr>
<tr>
<td>Standards may be enforced</td>
<td>Shifting standards, if any</td>
</tr>
</tbody>
</table>
Lecture No. 22
Software Change, Evolution and
Maintenance Part II

22.1 Software Maintenance

Software maintenance is the general process of changing a system after it has been diverted. The change may be simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancement to correct specification error or accommodate new requirements.

**Maintenance Examples**

**Y2K**
- Many, many systems had to be updated
- Language analyzers (find where changes need to be made)

**Anti-Virus Software**
- Don't usually have to update software, but must send virus definitions

**Operating System Patching**
- Microsoft, Apple, Linux/Unix
- OS is core to use of computer, so it must be constantly maintained

**Commercial Software in General**
- Customers need to be informed of updates
- Updates have to be easily available - web is good tool

22.2 The Maintenance Process

Maintenance process varies considerably depending on the types of software being maintained, the development processes used in an organization and people involved in the process.
22.3 Why is Maintenance Inefficient?

Factors adversely affect maintenance

– Lack of models or ignorance of available models (73%)
– Lack of documentation (67.6%)
– Lack of time to update existing documentation (54.1%)

Other factors (1994 study)

– Quality of original application
– Documentation quality
– Rotation of maintenance people

More factors (Yip ’95 study)

– Lack of human resources
– Different programming styles conflict
– Lack of documentation and tools
– Bad maintenance management
– Documentation policy
– Turnover

22.4 Maintenance Techniques

Architectural Evolution

There is a need to convert many legacy systems from a centralised architecture to a client-server architecture

Change drivers:

– Hardware costs. Servers are cheaper than mainframes
– User interface expectations. Users expect graphical user interfaces
– Distributed access to systems. Users wish to access the system from different, geographically
separated, computers

**User Interface Distribution**
- UI distribution takes advantage of the local processing power on PCs to implement a graphical user interface
- Where there is a clear separation between the UI and the application then the legacy system can be modified to distribute the UI
- Otherwise, screen management middleware can translate text interfaces to graphical interfaces

User Interface Distribution [SOM2004]

**22.5 The Management of Maintenance**

**Model of Maintenance Effort**
Model of maintenance effort $M = p + K^{(c-d)}$ [PRE2004]

- $M =$ total maintenance effort over entire lifecycle
- $p =$ productive efforts: analysis, design, code, test
- $c =$ complexity due to lack of structured design and documentation
- $d =$ degree of familiarization with the system
- $K =$ empirically determined constant

Model of maintenance effort $M = p + K^{(c-d)}$
- Cost of maintenance increases exponentially.
22.6 What Affects the Maintainability of an Application?

Application age
- (Software rust?) Older programs were probably worse written and have probably been patched more

Size
- Measured in KLOC, number of input/output files

Programming language
- 4gl's are supposed to produce more maintainable code than 3gl's

Processing environment
- Files harder to maintain than databases, real-time harder than batch

Analysis and design methodologies
- Well-designed software is supposed to be much easier to maintain

Structured programming
- There is conflicting evidence whether this really helps

Modularization
- (Central thesis of all the oo techniques) small reasonably self-contained pieces of code should be easier to maintain

Documentation generation
- Maintenance of documentation is as expensive as maintenance of code

End-user involvement
- Some researchers believe when end users are more involved maintenance decreases

Maintenance management
- Scheduling and the attitudes of management to affects productivity

22.7 Problems in Managing Maintenance

Changing priorities
- Chaotic nature of maintenance requests, the length of maintenance tasks causing new
requests to come along before an ongoing task is done.

Inadequate testing methods
- Lack of time set aside for testing, of comprehensive test data, of rigorous testing requirements as a standard for signing off.

Performance measurement difficulties
- How do you measure individual or group performance?

System documentation incomplete or non-existent
- Training takes a long time for learning an application so programmers get stuck on one piece of software.

Adapting to the rapidly changing business environment
- Hardware and software also become obsolete.

**Maintenance Prediction**

Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
- Change acceptance depends on the maintainability of the components affected by the change
- Implementing changes degrades the system and reduces its maintainability
- Maintenance costs depend on the number of changes and costs of change depend on maintainability
- Predicting the number of changes requires and understanding of the relationships between a system and its environment
- Tightly coupled systems require changes whenever the environment is changed
- Factors influencing this relationship are
  - Number and complexity of system interfaces
  - Number of inherently volatile system requirements
  - The business processes where the system is used
Lab No. 1 and 2
Introduction to Eclipse

What is Eclipse?
Eclipse started as a proprietary IBM product (IBM Visual age for Smalltalk/Java)
  – Embracing the open source model IBM opened the product up
Open Source
  – It is a general purpose open platform that facilitates and encourages the development of third
    party plug-ins
Best known as an Integrated Development Environment (IDE)
  – Provides tools for coding, building, running and debugging applications
Originally designed for Java, now supports many other languages
  – Good support for C, C++
  – Python, PHP, Ruby, etc…

Prerequisites for Running Eclipse
Eclipse is written in Java and will thus need an installed JRE or JDK in which to execute
  – JDK recommended

Obtaining Eclipse
Eclipse can be downloaded from…
  – Be sure to grab “Eclipse IDE for Java Developers”
Eclipse comes bundled as a zip file (Windows) or a tarball (all other operating systems)

Installing Eclipse
Simply unwrap the zip file to some directory where you want to store the executables
On windows
  – I typically unwrap the zip file to C:\eclipse\
  – I then typically create a shortcut on my desktop to the eclipse executable
  – C:\eclipse\eclipse.exe
Under Linux
I typically unwrap to /opt/eclipse/

Launching Eclipse
- Once you have the environment setup, go ahead and launch eclipse
- You should see the following splash screen…

Selecting a Workspace
- In Eclipse, all of your code will live under a workspace
- A workspace is nothing more than a location where we will store our source code and where Eclipse will write out our preferences
- Eclipse allows you to have multiple workspaces – each tailored in its own way
- Choose a location where you want to store your files, then click OK

Welcome to Eclipse
- The first time you launch Eclipse, you will be presented with a welcome screen
- From here you can access an overview to the platform, tutorials, sample code, etc…
- Click on the arrow on the right to get to the actual IDE
Eclipse IDE Components

Creating a New Project

- All code in Eclipse needs to live under a project
- To create a project: File → New → Java Project
Enter a name for the project, then click Finish.

The newly created project should then appear under the Package Explorer.
The src folder

- Eclipse automatically creates a folder to store your source code in called src

Creating a Class

- To create a class, simply click on the New button, then select Class
– This brings up the new class wizard
– From here you can specify the following...
  ● Package
  ● Class name
  ● Superclass
  ● Whether or not to include a main
  ● Etc…
– Fill in necessary information then click Finish to continue
The Created Class

As you can see a number of things have now happened...

Compiling Source Code

One huge feature of Eclipse is that it automatically compiles your code in the background

- You no longer need to go to the command prompt and compile code directly
- This means that errors can be corrected when made
- We all know that iterative development is the best approach to developing code, but going to
shell to do a compile can interrupt the normal course of development

- This prevents going to compile and being surprised with 100+ errors

**Example Compilation Error**

- This code contains a typo in the println statement…

- When clicking on the light bulb, Eclipse suggests changing println to either print or println
Running Code

An easy way to run code is to right click on the class and select Run As → Java Application
The output of running the code can be seen in the Console tab in the bottom pane.

**Run Configuration**

- Advanced options for executing a program can be found by right clicking the class then clicking Run As → Run…
Here you can change/add any of the following:

- JVM arguments
- Command line arguments
- Classpath settings
- Environment variables
- Which JVM to use
Re-Running Code
- After you run the code a first time, you can re-run it just by selecting it from the run drop down menu

Debugging Code
- Eclipse comes with a pretty good built-in debugger
- You can set break points in your code by double clicking in the left hand margin – break points are represented by these blue bubbles
An easy way to enter debug mode is to right click on the class and select Debug As → Java Application.

The first time you try to debug code you will be presented with the following dialog.
Eclipse is asking if you want to switch to a perspective that is more suited for debugging, click Yes.

Eclipse has many perspectives based on what you are doing (by default we get the Java perspective).

**Debug Perspective**

- These buttons allow you to step through the code.
- Variables in scope are listed here along with their current values. By right clicking you can change values of variables as your program is running.
- This pane shows the current line of code we broke on.
- Output console, just like in normal run mode.
- Current high level location (class and method).
- Note new Debug perspective – click Java to return to normal.
Sampling of Some Other Features

- Import organization
- Context assist
- Javadoc assist
- Getter/Setter generation
- Add unimplemented methods
- Exception handling
- Reminders
- Local history

Import Organization
- Eclipse can automatically include import statements for any classes you are using, just press Control + Shift + o (letter o)
– If the class is ambiguous (more than one in the API) then it will ask you to select the correct one

- Import statements automatically included and organized
  - You can organize imports to clean them up at any time
Context Assist

- If you are typing and press a “.” character and pause a second, Eclipse will show you a list of all available methods for the class
  - Prevents having to browse javadocs to see what methods are available
  - Get context assist at any time by pressing Control + Space
Javadoc Assist

- Eclipse can also help generate javadoc comments for you, simply place the cursor before the method and then type `/**` then Enter

- Eclipse will automatically generate a javadoc header for the method all stubbed out with the parameters, return type and exceptions
Getter/Setter Generation
- Eclipse can automatically generate getters and setters for member of a class...
- To generate getters and setters, right click in the main pane, then select Source → Generate Getters and Setters

- Here you can selectively choose members for which to generate getters and setters
Eclipse will then automatically generate the code for the getters and setters

Add Unimplemented Methods

Eclipse can also stub out methods that need to be present as a result of implementing an interface…
– You can use the quick fix light bulb to add the interfaces unimplemented methods to the class

– Again Eclipse will go ahead and stub out the method for us
Exception Handling

- Eclipse will also pick up on unhandled exceptions

- By clicking on the quick fix light bulb, Eclipse can suggest what to do to handle the exception
- Eclipse can automatically add a “throws declaration” to the method signature.

- Alternately, Eclipse can also wrap the code inside a try/catch block.
Tasks

- Eclipse allows you to insert reminders into your code and stores them for you to come back and revisit them

- Eclipse recognizes the following tags inside comments...
  - TODO
  - FIXME
  - XXX

- You can even add your own custom tasks through the preferences menu

  • To add a table of all reminders in all of your source code you can add the Tasks view by clicking on Window → Show View → Tasks
This neatly displays all tasks in a tabular form
Local History

- Eclipse maintains a local history of file revisions which can be accessed by right clicking on the class, then selecting Compare With → Local History…

- Previous saved revisions are displayed in the History pane, double click a revision to view in the built-in diff viewer

Summary
Benefits
- Code completion
- Faster code/compile/run cycles (real time)
- Open source (free)
- Extensible (plugins)

Disadvantages
- Pretty heavyweight
- Requires JRE
- Learning Curve

Lab No. 3
Classes and Objects in Java
Basics of Classes in Java

Classes
A class is a collection of fields (data) and methods (procedure or function) that operate on that data.

A class is a collection of fields (data) and methods (procedure or function) that operate on that data. The basic syntax for a class definition:

class ClassName [extends SuperClassName]
{
[fields declaration]
[methods declaration]

– Bare bone class – no fields, no methods

public class Circle {
    // my circle class
}

**Adding Fields: Class Circle with fields**

– Add fields

public class Circle {
    public double x, y;  // centre coordinate
    public double r;     // radius of the circle
}

– The fields (data) are also called the *instance* variables.

**Adding Methods**

– A class with only data fields has no life. Objects created by such a class cannot respond to any messages.

– Methods are declared inside the body of the class but immediately after the declaration of data fields.

– The general form of a method declaration is:

```
type MethodName (parameter-list)
{
    Method-body;
}
```

**Adding Methods to Class Circle**

public class Circle {
    public double x, y; // centre of the circle
    public double r;    // radius of circle

    // Methods to return circumference and area
    public double circumference() {
        return 2*3.14*r;
    }
}
}  
public double area() {
    return 3.14 * r * r;
}
}  

Data Abstraction  
– Declare the Circle class, have created a new data type – Data Abstraction  
– Can define variables (objects) of that type: 

Circle aCircle;  
Circle bCircle;  
aCircle, bCircle simply refers to a Circle object, not an object itself.  

Creating objects of a class  
– Objects are created dynamically using the new keyword.  
– aCircle and bCircle refer to Circle objects  

aCircle = new Circle();  
bCircle = new Circle();
Creating objects of a class

aCircle = new Circle();
bCircle = new Circle();
bCircle = aCircle;

---

Automatic garbage collection

- The object does not have a reference and cannot be used in future.
- The object becomes a candidate for automatic garbage collection.
- Java automatically collects garbage periodically and releases the memory used to be used in the future.

Accessing Object/Circle Data

- Similar to C syntax for accessing data defined in a structure.

ObjectName. VariableName
ObjectName.MethodName(parameter-list)

Circle aCircle = new Circle();
aCircle.x = 2.0 // initialize center and radius
aCircle.y = 2.0
aCircle.r = 1.0

Executing Methods in Object/Circle

- Using Object Methods:
Using Circle Class

// Circle.java: Contains both Circle class and its user class
// Add Circle class code here

class MyMain
{

    public static void main(String args[])
    {
        Circle aCircle; // creating reference
        aCircle = new Circle(); // creating object
        aCircle.x = 10; // assigning value to data field
        aCircle.y = 20;
        aCircle.r = 5;
        double area = aCircle.area(); // invoking method
        double circumf = aCircle.circumference();
        System.out.println("Radius="+aCircle.r+" Area="+area);
        System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
    }
}

[raj@mundroo]%: java MyMain
Radius=5.0 Area=78.5
Radius=5.0 Circumference =31.400000000000002

Refer to the Earlier Circle Program

// Circle.java: Contains both Circle class and its user class
//Add Circle class code here

class MyMain
{
    public static void main(String args[])
    {
        Circle aCircle;  // creating reference
        aCircle = new Circle(); // creating object
        aCircle.x = 10;  // assigning value to data field
        aCircle.y = 20;
        aCircle.r = 5;
        double area = aCircle.area(); // invoking method
        double circumf = aCircle.circumference();
        System.out.println("Radius="+aCircle.r+" Area="+area);
        System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
    }
}

Better way of Initialising or Access Data Members x, y, r

- When there too many items to update/access and also to develop a readable code, generally it is done by defining specific method for each purpose.
- To initialise/Update a value:
  - aCircle.setX( 10 )
- To access a value:
  - aCircle.getX()
- These methods are informally called as Accessors or Setters/Getters Methods.

Accessors – “Getters/Setters”

    public class Circle {
        public double x,y,r;
        //Methods to return circumference and area
        public double getX() { return x;}
        public double getY() { return y;}
    }
public double getR() { return r;}
public double setX(double x_in) { x = x_in;}
public double serY(double y_in) { y = y_in;}
public double setR(double r_in) { r = r_in;}

How does this code looks? More readable?

// Circle.java: Contains both Circle class and its user class
// Add Circle class code here
class MyMain {
    public static void main(String args[]) {
        Circle aCircle; // creating reference
        aCircle = new Circle(); // creating object
        aCircle.setX(10);
        aCircle.setY(20);
        aCircle.setR(5);
        double area = aCircle.area(); // invoking method
        double circumf = aCircle.circumference();
        System.out.println("Radius="+aCircle.getR()+" Area="+area);
        System.out.println("Radius="+aCircle.getR()+" Circumference ="+circumf);
    }
}

Object Initialization
- When objects are created, the initial value of data fields is unknown unless its users explicitly do so. For example,
  - ObjectName.DataField1 = 0; // OR
  - ObjectName.SetDataField1(0);
- In many cases, it makes sense if this initialisation can be carried out by default without the users explicitly initialising them.
For example, if you create an object of the class called “Counter”, it is natural to assume that the counter record-keeping field is initialised to zero unless otherwise specified differently.

```java
class Counter {
    int CounterIndex;
    ...
}
```

Counter counter1 = new Counter();

- What is the value of “counter1.CounterIndex”?
- In Java, this can be achieved through a mechanism called constructors.

**What is a Constructor?**

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

**Defining a Constructor**

- Like any other method

```java
public class ClassName {
    // Data Fields...

    // Constructor
    public ClassName() {
    }
}
```
// Method Body Statements initialising Data Fields


//Methods to manipulate data fields


– Invoking:

There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.

**Defining a Constructor: Example**

```java
class Counter {
    int CounterIndex;
    // Constructor
    public Counter()
    {
        CounterIndex = 0;
    }
    //Methods to update or access counter
    public void increase()
    {
        CounterIndex = CounterIndex + 1;
    }
    public void decrease()
    {
        CounterIndex = CounterIndex - 1;
    }
    int getCounterIndex()
    {
        return CounterIndex;
    }
}
```
Trace counter value at each statement and what is the output?

class MyClass {
    public static void main(String args[]) {
        Counter counter1 = new Counter();
        counter1.increase();
        int a = counter1.getCounterIndex();
        counter1.increase();
        int b = counter1.getCounterIndex();
        if (a > b)
            counter1.increase();
        else
            counter1.decrease();
        System.out.println(counter1.getCounterIndex());
    }
}

Lab No 4
Classes and Objects in Java
Basics of Classes in Java

Defining a Constructor: Example

public class Counter {
    int CounterIndex;
    // Constructor
    public Counter() {
        CounterIndex = 0;
    }
    //Methods to update or access counter
    public void increase()
{  
    CounterIndex = CounterIndex + 1;
}

class MyClass {
    public static void main(String args[]) {
        Counter counter1 = new Counter();
        counter1.increase();
        int a = counter1.getCounterIndex();
        counter1.increase();
        int b = counter1.getCounterIndex();
        if (a > b) {
            counter1.increase();
        } else {
            counter1.decrease();
            System.out.println(counter1.getCounterIndex());
        }
    }
}

A Counter with User Supplied Initial Value?
– This can be done by adding another constructor method to the class.

public class Counter {
    int CounterIndex;
    // Constructor 1
public Counter()
{
    CounterIndex = 0;
}
public Counter(int InitValue )
{
    CounterIndex = InitValue;
}

// A New User Class: Utilising both constructors
Counter counter1 = new Counter();
Counter counter2 = new Counter (10);

Adding a Multiple-Parameters Constructor to our Circle Class

public class Circle {
    public double x,y,r;
    // Constructor
    public Circle(double centreX, double centreY, double radius)
    {
        x = centreX;
        y = centreY;
        r = radius;
    }
    //Methods to return circumference and area
    public double circumference() { return 2*3.14*r; }
    public double area() { return 3.14 * r * r; }
}

Constructors initialise Objects

– Recall the following OLD Code Segment:

Circle aCircle = new Circle();
aCircle.x = 10.0; // initialize center and radius
aCircle.y = 20.0
aCircle.r = 5.0;
At creation time the centre and radius are not defined. These values are explicitly set later.

Constructors initialise Objects

- With defined constructor

```java
Circle aCircle = new Circle(10.0, 20.0, 5.0);
```

- Sometimes want to initialize in a number of different ways, depending on circumstance.
- This can be supported by having multiple constructors having different input arguments.

```java
public class Circle {
    public double x, y, r; // instance variables

    // Constructors
    public Circle(double centreX, double centreY, double radius) {
        x = centreX; y = centreY; r = radius;
    }
    public Circle(double radius) { x = 0; y = 0; r = radius; }
    public Circle() { x = 0; y = 0; r = 1.0; }

    // Methods to return circumference and area
    public double circumference() { return 2*3.14*r; }
    public double area() { return 3.14 * r * r; }
}
```
Initializing with constructors

```java
public class TestCircles {
    public static void main(String args[]) {
        Circle circleA = new Circle(10.0, 12.0, 20.0);
        Circle circleB = new Circle(10.0);
        Circle circleC = new Circle();
    }
}
```

Method Overloading

- Constructors all have the same name.
- Methods are distinguished by their signature:
  - Name
  - Number of arguments
  - Type of arguments
  - Position of arguments
- That means, a class can also have multiple usual methods with the same name.
- Not to confuse with method overriding (coming up), method overloading:

Polymorphism

- Allows a single method or operator associated with different meaning depending on the type of data passed to it
- It can be realized through:
  - Method Overloading
  - Operator Overloading (Supported in C++, but not in Java)
- Defining the same method with different argument types (method overloading) - polymorphism.
The method body can have different logic depending on the data type of arguments.

**Scenario**

- A Program needs to find a maximum of two numbers or Strings. Write a separate function for each operation.

- In C:
  - int max_int(int a, int b)
  - int max_string(char *s1, char *s2)
  - max_int (10, 5) or max_string ("melbourne", "sydney")

- In Java:
  - int max(int a, int b)
  - int max(String s1, String s2)
  - max(10, 5) or max("melbourne", "sydney")

**A Program with Method Overloading**

// Compare.java: a class comparing different items

class Compare {
    static int max(int a, int b) {
        if( a > b)
            return a;
        else
            return b;
    }
    static String max(String a, String b) {
        if( a.compareTo (b) > 0)
            return a;
        else
            return b;
    }
    public static void main(String args[])
{  
    String s1 = "Melbourne";
    String s2 = "Sydney";
    String s3 = "Adelaide";
    int a = 10;
    int b = 20;
    System.out.println(max(a, b)); // which number is big
    System.out.println(max(s1, s2)); // which city is big
    System.out.println(max(s1, s3)); // which city is big
}

The New this keyword

- this keyword can be used to refer to the object itself. It is generally used for accessing class members (from its own methods) when they have the same name as those passed as arguments.

class Circle {

    public double x,y,r;

    // Constructor
    public Circle (double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    //Methods to return circumference and area

}
Static Members
- Java supports definition of global methods and variables that can be accessed without creating objects of a class. Such members are called Static members.
- Define a variable by marking with the static methods.
- This feature is useful when we want to create a variable common to all instances of a class.
- One of the most common examples is to have a variable that could keep a count of how many objects of a class have been created.
- Note: Java creates only one copy for a static variable which can be used even if the class is never instantiated.

Static Variables
- Define using static:
public class Circle {
    // class variable, one for the Circle class, how many circles
    public static int numCircles;
    // instance variables, one for each instance of a Circle
    public double x,y,r;
    // Constructors...
}
- Access with the class name (ClassName StatVarName):
nCircles = Circle.numCircles;

Static Variables - Example
- Using static variables:
public class Circle {
    // class variable, one for the Circle class, how many circles
    private static int numCircles = 0;
    private double x,y,r;
    // Constructors...
    Circle (double x, double y, double r) {
        this.x = x;
        this.y = y;
}
```java
    this.r = r;
    numCircles++;

    }
}

Class Variables - Example

- Using static variables:

public class CountCircles {
    public static void main(String args[]){
        Circle circleA = new Circle(10, 12, 20); // numCircles = 1
        Circle circleB = new Circle(5, 3, 10); // numCircles = 2
    }
}

Instance Vs Static Variables

- Instance variables: One copy per object. Every object has its own instance variable.
  E.g. x, y, r (centre and radius in the circle)

- Static variables: One copy per class.
  E.g. numCircles (total number of circle objects created)

Static Methods

- A class can have methods that are defined as static (e.g., main method).
- Static methods can be accessed without using objects. Also, there is NO need to create objects.
- They are prefixed with keyword “static”
- Static methods are generally used to group related library functions that don’t depend on data
```
members of its class. For example, Math library functions.

**Comparator class with Static methods**

// Comparator.java: A class with static data items comparision methods
class Comparator {
    public static int max(int a, int b) {
        if (a > b)
            return a;
        else
            return b;
    }

    public static String max(String a, String b) {
        if (a.compareTo(b) > 0)
            return a;
        else
            return b;
    }
}

class MyClass {
    public static void main(String args[]) {
        String s1 = "Melbourne";
        String s2 = "Sydney";
        String s3 = "Adelaide";
        int a = 10;
        int b = 20;
        System.out.println(Comparator.max(a, b)); // which number is big
        System.out.println(Comparator.max(s1, s2)); // which city is big
        System.out.println(Comparator.max(s1, s3)); // which city is big
    }
}

**Static methods restrictions**

- They can only call other static methods.
- They can only access static data.
- They cannot refer to “this” or “super” (more later) in anyway.
Lab 05
Abstraction, Inheritance, and Polymorphism
in Java

Object Orientation
“Object Orientation involving encapsulation, inheritance, polymorphism, and abstraction, is an important approach in programming and program design. It is widely accepted and used in industry and is growing in popularity in the first and second college-level programming courses.”

Some other reasons to move on to Java:
– Platform-independent software
– Relatively easy graphics and GUI programming
– Lots of library packages
– Free compiler and IDEs
– Colleges are teaching it
– Companies are using it
– Students want it
– (Teachers welcome it... ;)
– (Programmers drink it... :)

What are OOP’s claims to fame?
– Better suited for team development
– Facilitates utilizing and creating reusable software components
– Easier GUI programming
– Easier program maintenance

OOP in a Nutshell:
– A program models a world of interacting objects
– Objects create other objects and “send messages” to each other (in Java, call each other’s methods)
– Each object belongs to a class; a class defines properties of its objects
– A class implements an ADT; the data type of an object is its class
Programmers write classes (and reuse existing classes)

Case Study: Dance Studio

Good news: The classes are fairly short
In OOP, the number of classes is not considered a problem

**Abstraction**

Abstraction means ignoring irrelevant features, properties, or functions and emphasizing the relevant ones (relevant to the given project (with an eye to future reuse in similar projects)).

**Encapsulation**

Encapsulation means that all data members (fields) of a class are declared private. Some methods may be private, too.

The class interacts with other classes (called the clients of this class) only through the class’s constructors and public methods. Constructors and public methods of a class serve as the interface to class’s clients.

```java
public abstract class Foot
{
    private static final int footWidth = 24;
    private boolean amLeft;
    private int myX, myY;
    private int myDir;
    private boolean myWeight;
    // Constructor:
    protected Foot(String side, int x, int y, int dir)
    {
        amLeft = side.equals("left");
        myX = x;
        myY = y;
        myDir = dir;
    }
}```
myWeight = true;
}

Encapsulation ensures that structural changes remain local
– Changes in the code create software maintenance problems
– Usually, the structure of a class (as defined by its fields) changes more often than the class’s
  constructors and methods
– Encapsulation ensures that when fields change, no changes are needed in other classes (a
  principle known as “locality”)

Inheritance
– A class can extend another class, inheriting all its data members and methods while
  redefining some of them and/or adding its own.
– Inheritance represents there is a relationship between data types. For example: a
  FemaleDancer is a Dancer.

Inheritance Terminology:

```
subclass extends superclass

or

derived class extends base class
```

Constructors are not inherited. The FemaleDancer class only adds a constructor:

```
public class FemaleDancer extends Dancer
{
    public FemaleDancer(String steps[ ],
                            int x, int y, int dir)
    {
```

leftFoot = new FemaleFoot("left", x, y, dir);
rightFoot = new FemaleFoot("right", x, y, dir);
leftFoot.move(-Foot.getWidth() / 2, 0);
rightFoot.move(Foot.getWidth() / 2, 0);

Inheritance may be used to define a hierarchy of classes in an application:
All methods of the base library class are available in your derived class. You don’t need to have the source code of a class to extend it

**Polymorphism**

- Polymorphism ensures that the appropriate method is called for an object of a specific type when the object is disguised as a more general type.
- Good news: polymorphism is already supported in Java — all you have to do is use it properly.

**Situation 1:**

- A collection (array, list, etc.) contains objects of different but related types, all derived from the same common base class.
- Polymorphism replaces old-fashioned use of explicit object attributes and if-else (or switch) statements, as in:

```java
public abstract class Foot {
    ...

    public void draw(Graphics g) {
        ...
        if (isLeft())
            drawLeft(g);
        else
```

This diagram illustrates the hierarchy of classes involved in the example:

```
Object
  ↓
Foot
  ↓
MaleFoot
  ↓
MaleLeftFoot
  ↓
MaleRightFoot

FemaleFoot
  ↓
FemaleLeftFoot
  ↓
FemaleRightFoot
```
Lab 6

GUI Building with the AWT in Java

AWT (Abstract Window Toolkit)
- Present in all Java implementations
- Described in most Java textbooks
- Adequate for many applications
- Uses the controls defined by your OS
- therefore it's “least common denominator”
- Difficult to build an attractive GUI
- import java.awt.*;
  import java.awt.event.*;

Swing
- Same concepts as AWT
- Doesn’t work in ancient Java implementations (Java 1.1 and earlier)
- Many more controls, and they are more flexible
- Some controls, but not all, are a lot more complicated
- Gives a choice of “look and feel” packages
- Much easier to build an attractive GUI
- import javax.swing.*;

Swing vs. AWT
- Swing is bigger, slower, and more complicated
- But not as slow as it used to be
- Swing is more flexible and better looking
Swing and AWT are incompatible—you can use either, but you can’t mix them.
Actually, you can, but it’s tricky and not worth doing.
Learning the AWT is a good start on learning Swing.
Many of the most common controls are just renamed.
AWT: Button b = new Button("OK");
Swing: JButton b = new JButton("OK");

To build a GUI...
- Make somewhere to display things—usually a Frame or Dialog (for an application), or an Applet
- Create some Components, such as buttons, text areas, panels, etc.
- Add your Components to your display area
- Arrange, or lay out, your Components
- Attach Listeners to your Components
- Interacting with a Component causes an Event to occur
- A Listener gets a message when an interesting event occurs, and executes some code to deal with it

Containers and Components
- The job of a Container is to hold and display Components
- Some common subclasses of Component are Button, Checkbox, Label, Scrollbar, TextField, and TextArea
- A Container is also a Component
- This allows Containers to be nested
- Some Container subclasses are Panel (and Applet), Window, and Frame

An Applet is Panel is a Container

```
java.lang.Object
    
    +-----java.awt.Component
    
    +-----java.awt.Container
```
…so you can display things in an Applet

**Example: A "Life" applet**

An application has a public static void main (String args[ ]) method, but an Applet usually does not

- An Applet's main method is in the Browser
- To write an Applet, you extend Applet and override some of its methods
- The most important methods are init( ), start( ), and paint(Graphics g)

**To create an applet**

- Public class myapplet extends Applet { … }
- This is the *only* way to make an Applet
- You can add components to the applet
– The best place to add components is in init( )
– You can paint directly on the applet, but…
– …it’s better to paint on a contained component
– Do all painting from paint(Graphics g)

Some types of components

Creating components

Label lab = new Label("Hi, Dave!");

Button but = new Button("Click me!");

Checkbox toggle = new Checkbox("toggle");

TextField txt =
    new TextField("Initial text.", 20);

Scrollbar scrolly = new Scrollbar
    (Scrollbar.HORIZONTAL, initialValue,
    bubbleSize, min Value, max Value);

Adding components to the Applet

class MyApplet extends Applet {

```java
...
public void init () {

    add (lab); // same as this.add(lab)

    add (but);

    add (toggle);

    add (txt);

    add (scrolly);

    ...

Creating a Frame

When you create an Applet, you get a Panel “for free”

When you write a GUI for an application, you need to create and use a Frame:

– Frame frame = new Frame();
– frame.setTitle("My Frame");
– frame.setSize(300, 200); // width, height
– ... add components ...
– frame.setVisible(true);

Or:

– class MyClass extends Frame {
...
– setTitle("My Frame"); // in some instance method

Arranging components

– Every Container has a layout manager
– The default layout for a Panel is FlowLayout
– An Applet is a Panel
– Therefore, the default layout for a Applet is FlowLayout
– You could set it explicitly with
    setLayout (new FlowLayout( ));
– You could change it to some other layout manager
FlowLayout

– Use add(component); to add to a component when using a FlowLayout
– Components are added left-to-right
– If no room, a new row is started
– Exact layout depends on size of Applet
– Components are made as small as possible
– FlowLayout is convenient but often ugly

**Complete example: FlowLayout**

```java
class FlowLayoutExample extends Applet {

public void init() {

setLayout (new FlowLayout()); // default
add (new Button("One"));
add (new Button("Two"));
add (new Button("Three"));
add (new Button("Four"));
add (new Button("Five"));
add (new Button("Six"));
}
}
```

BorderLayout

– At most five components can be added
– If you want more components, add a Panel, then add components to it.
– setLayout (new borderlayout());
– add (new Button("NORTH"), BorderLayout.NORTH);

**BorderLayout with five Buttons**

```java
class BorderLayoutExample extends Applet {

public void init() {

```
setLayout (new BorderLayout ());
add (new Button ("NORTH"), BorderLayout.NORTH);
add (new Button ("SOUTH"), BorderLayout.SOUTH);
add (new Button ("EAST"), BorderLayout.EAST);
add (new Button ("WEST"), BorderLayout.WEST);
add (new Button ("CENTER"), BorderLayout.CENTER);
}

Complete example: BorderLayout

import java.awt.*;
import java.applet.);

public class BorderLayoutExample extends Applet {
    public void init () {
        setLayout (new BorderLayout());
        add(new Button("One"), BorderLayout.NORTH);
        add(new Button("Two"), BorderLayout.WEST);
        add(new Button("Three"), BorderLayout.CENTER);
        add(new Button("Four"), BorderLayout.EAST);
        add(new Button("Five"), BorderLayout.SOUTH);
        add(new Button("Six"), BorderLayout.SOUTH);
    }
}

Using a Panel

    Panel p = new Panel();
    add (p, BorderLayout.SOUTH);
    p.add (new Button ("Button 1"));
p.add (new Button ("Button 2"));

**GridLayout**

- The GridLayout manager divides the container up into a given number of rows and columns:
  - New `gridlayout(rows, columns)`
    All sections of the grid are equally sized and as large as possible

**Complete example: GridLayout**

```java
import java.awt.*;
import java.applet.*;

public class GridLayoutExample extends Applet {
    public void init () {
        setLayout(new GridLayout(2, 3));
        add(new Button("One"));
        add(new Button("Two"));
        add(new Button("Three"));
        add(new Button("Four"));
        add(new Button("Five"));
    }
}
```

**Making components active**

- Most components already *appear* to do something--buttons click, text appears
- To associate an action with a component, attach a *listener* to it
- Components send events, listeners listen for events
- Different components may send different events, and require different listeners

**Listeners**

- Listeners are interfaces, not classes
  ```java
  class MyButtonListener implements ActionListener {
  }
  ```
- An interface is a group of methods that *must* be supplied
- When you say implements, you are *promising* to supply those methods
Writing a Listener

For a Button, you need an ActionListener

```java
b1.addActionListener
        (new MyButtonListener ( ));
```

An ActionListener must have an actionPerformed(ActionEvent) method

```java
public void actionPerformed(ActionEvent e) {
...
}
```

**MyButtonListener**

```java
public void init () {
...

    b1.addActionListener (new MyButtonListener ());
}
```

class MyButtonListener implements ActionListener {

```java
public void actionPerformed (.ActionEvent e) {
    showStatus ("Ouch!");
}
```
}

Listeners for TextFields

- An ActionListener listens for someone hitting the Enter key
- An ActionListener requires this method:

```java
public void actionPerformed (ActionEvent e) {
```
- You can use getText( ) to get the text

```java
    A TextListener listens for any and all keys
```
- A TextListener requires this method:

```java
    public void textValueChanged(TextEvent e)
```
AWT and Swing

- AWT Buttons vs. Swing JButton:
  - A Button is a Component
  - A JButton is an AbstractButton, which is a JComponent, which is a Container, which is a Component

Containers:

- Swing uses AWT Containers
- AWT Frames vs. Swing JFrame:
  - A Frame is a Window is a Container is a Component
  - A JFrame is a Frame, etc.

Layout managers:

- Swing uses the AWT layout managers, plus a couple of its own

Listeners:

- Swing uses many of the AWT listeners, plus a couple of its own

- Bottom line: Not only is there a lot of similarity between AWT and Swing, but Swing actually uses much of the AWT
Simplest GUI programming: JOptionPane

An option pane is a simple dialog box for graphical input/output

**Advantages:**
- Simple
- Flexible (in some ways)
- Looks better than the black box of death

**Disadvantages:**
- Created with static methods; not very object-oriented
- Not very powerful (just simple dialog boxes)

**Types of JOptionPane**
- Public static void showMessageDialog(Component parent, Object message) Displays a message on a dialog with an OK button.
- Public static int showConfirmDialog(Component parent, Object message) Displays a message and list of choices Yes, No, Cancel
- Public static String showInputDialog(Component parent, Object message) Displays a message and text field for input, and returns the value entered as a String.

**JOptionPane examples 1**

ShowMessageDialog analogous to System.out.println for displaying a simple message

```java
import javax.swing.*;

class MessageDialogExample {
    public static void main(String[] args) {
        JOptionPane.showMessageDialog(null, "How's the weather?");
        JOptionPane.showMessageDialog(null,
```

JOptionPane examples 2
showConfirmDialog analogous to a System.out.print that prints a question, then reading an input value from the user (can only be one of the provided choices)
import javax.swing.*;
class ConfirmDialogExample {
    public static void main(String[] args) {
        int choice = JOptionPane.showConfirmDialog(null,
            "Erase your hard disk?");
        if (choice == JOptionPane.YES_OPTION) {
            JOptionPane.showMessageDialog(null, "Disk erased!");
        } else {
            JOptionPane.showMessageDialog(null, "Cancelled.");
        }
    }
}

JOptionPane examples 3
ShowInputDialog analogous to a System.out.print that prints a question, then reading an input value from the user (can be any value)
import javax.swing.*;
class InputDialogExample {
    public static void main(String[] args) {
        String name = JOptionPane.showInputDialog(null,
            "What's yer name, pardner?");
        JOptionPane.showMessageDialog(null, "Yeehaw, " + name);
    }
}
Onscreen GUI elements

- **Windows**: actual first-class citizens of desktop; also called top-level containers
  examples: frame, dialog box

- **Components**: gui widgets
  examples: button, text box, label

- **Containers**: logical grouping for components
  example: panel

**Java GUI: AWT and Swing**

- Sun's initial idea: create a set of classes/methods that can be used to write a multi-platform GUI (Abstract Windowing Toolkit, or AWT)
  Problem: not powerful enough; limited; a bit clunky to use

- Second edition (JDK v1.2): Swing
  A newer library written from the ground up that allows much more powerful graphics and GUI construction

- Drawback: Both exist in Java now; easy to get them mixed up; still have to use both sometimes!

**Swing component hierarchy**

```
java.lang.Object
 +--java.awt.Component
    +--java.awt.Container
         |   
         |   +--javax.swing.JComponent
         |       |   +--javax.swing.JButton
         |       |   +--javax.swing.JLabel
         |       |   +--javax.swing.JMenuBar
         |       |   +--javax.swing.JOptionPane
         |       |   +--javax.swing.JPanel
         |       |   +--javax.swing JTextArea
         |       |   +--javax.swing.JTextField
```
import java.awt.*;
import javax.swing.);

Methods of all Swing components

– Public int getWidth()
  public int getHeight()
  Allow access to the component's current width and height in pixels.

– Public boolean isEnabled()
  Returns whether the component is enabled (can be interacted with).

– Public boolean isVisible()
  Returns whether the component is visible (can be seen on the screen).

More JComponent methods

– Public void setBackground(Color c)
  Sets the background color of the component to be the given color.

– Public void setFont(Font f)
  Sets the font of the text on the given component to be the given font.

– Public void setEnabled(boolean b)
  Sets whether the component is enabled (can be interacted with).

– Public void setVisible(boolean b)
  Sets whether the component is visible (can be seen on the screen). Set to true to show the component, or set to false to hide the component.

JFrame

A frame is a graphical window that can be used to hold other components

– Public JFrame() or public JFrame(String title)
  Creates a frame with an optional title.

– Public void setTitle(String text)
  Puts the given text in the frame’s title bar.

– Public void setDefaultCloseOperation(int op)
Makes the frame perform the given action when it closes. Common value:
JFrame.EXIT_ON_CLOSE

– Public void add(Component comp)
  Places the given component or container inside the frame.
– How would we add more than one component to the frame?
– Public void pack()
  Resizes the frame to fit the components inside it.
– NOTE: Call setVisible(true) to make a frame appear on the screen after creating it.

Lab 08
Java Programming with SWING
GUI Builder

JButton, JLabel

– The most common component a button is a clickable onscreen region that the user interacts with to perform a single command
– A text label is simply a string of text displayed on screen in a graphical program. Labels often give information or describe other components
– Public JButton(String text)
  public JLabel(String text)
  Creates a new button / label with the given string as its text.
– Public String getText()
  Returns the text showing on the button / label.
– Public void setText(String text)
  Sets button / label's text to be the given string.

JTextField, JTextArea

– A text field is like a label, except that the text in it can be edited and modified by the user.
  Text fields are commonly used for user input, where the user types information in the field
and the program reads it

A text area is a multi-line text field

- public JTextField(int columns)
- public JTextArea(int lines, int columns)
  Creates a new text field that is the given number of columns (letters) wide.
- public String getText()
  Returns the text currently in the field.
- public void setText(String text)
  Sets field's text to be the given string.

**JCheckBox, JRadioButton**

- A check box is a toggleable button with two states: checked and unchecked
- A radio button is a button that can be selected; usually part of a group of mutually-exclusive radio buttons (1 selectable at a time)
- public JCheckBox / JRadioButton(String text)
  public JCheckBox(String text, boolean isChecked)
  Creates checked/unchecked check box with given text.
- public boolean isSelected()
  Returns true if check box is checked.
- public void setSelected(boolean selected)
  Sets box to be checked/unchecked.

**ButtonGroup**

A logical group of radio buttons that ensures that only one is selected at a time

- public ButtonGroup()
- public void add(JRadioButton button)

The ButtonGroup is not a graphical component, just a logical group; the RadioButtons themselves are added to the container, not the ButtonGroup

**Icon/ImageIcon**

Allows you to put a picture on a button, label or other component

Public class ImageIcon implements Icon

- Public imageicon(String filename)
In JButton, JRadioButton, JCheckBox, JLabel, etc...

- Constructor that takes an Icon
- Public void setIcon(Icon)
- Public void setSelectedIcon(Icon)
- Public void setRolloverIcon(Icon)

**JScrollPane**

A special container that holds a component, using scrollbars to allow that component to be seen

- Public JScrollPane(Component comp)

Wraps the given component with scrollbars.

After constructing the scroll pane, add the scroll pane to the container, not the original component.

```java
contentPane.add(new JScrollPane(textarea),
        BorderLayout.CENTER);
```

**JFileChooser**

A special dialog box that allows the user to select one or more files/folders

- public JFileChooser()
- public JFileChooser(String currentDir)
- public int showOpenDialog(Component parent)
- public int showSaveDialog(Component parent)
- public File getSelectedFile()
- public static int APPROVE_OPTION, CANCEL_OPTION

Possible result values from showXxxDialog(..).

```java
FileChooser chooser = new JFileChooser();
int result = chooser.showSaveDialog(this);
if (result == JFileChooser.APPROVE_OPTION)
    this.saveData(chooser.getSelectedFile().getName());
```

**JColorChooser**

Another special dialog that lets the user pick from a palette of colors
– public JColorChooser()
– public JColorChooser(Color initial)
– public Color showDialog(Component parent,
  String title, Color initialColor)
  returns null if user chose Cancel option

**JMenuBar**
The top-level container that holds menus; can be attached to a frame
– public JMenuBar()
– public void add(JMenu menu)

Usage: in JFrame, the following method exists:
– public void setJMenuBar(JMenuBar bar)

**JMenu**
A menu to hold menu items; menus can contain other menus (Composite)
– public JMenu(String text)
– public void add(JMenuItem item)
– public void addSeparator()
– public void setMnemonic(int mnemonic)

**JMenuItem**
An entry in a frame's Menu bar, which can be clicked to perform commands
– Public JMenuItem(String text)
– Public JMenuItem(String text, Icon icon)
– Public JMenuItem(String text, int mnemonic)
– Public void addActionListener(
  actionlistener al)
– Public void setAccelerator(keystroke ks)
– Public void setEnabled(boolean b)
– Public void setMnemonic(int mnemonic)

**JCheckBoxMenuItem / JRadioButtonMenuItem**
Radio button and checkbox-like menu items
– Public J menuitem(String text)
– Public J menuitem(String text, boolean select)
– Public J menuitem(String text, Icon icon)
– Public J menuitem(String text,
    Icon icon, boolean selected)
– Public void addactionlistener(actionlistener al)
– Public boolean isselected()
– Public void setselected(boolean b)

Recall: in a ButtonGroup, the following method exists:
– Public void add(abstractbutton button)

These two classes extend AbstractButton!

**Mnemonics**

Menu hotkey assigned to a button or other graphical component usually visible as an underlined key, activated by pressing Ctrl+key (buttons) or Alt+key (menus)
– Only work when input focus is on the appropriate component (affects menus)
– Usage: call setmnemonic(char) method
– Menu items also have constructor that takes mnemonic

myQuitButton.setMnemonic('Q');
JMenuItem myNewItem = new JMenuItem("New", 'N');
// or,
myNewItem.setMnemonic('N');

**Accelerators**

Global hotkey combination that performs an action (ex: Alt-X to exit program) even on components that aren't in focus / visible
– Can be run at any time in the application
– Can optionally include modifiers like shift, alt
– Created by calling the getkeystroke method of the keystroke class, and passing this to setaccelerator method of various components (buttons, menus)

menuItem.setAccelerator(
KeyStroke.getKeyStroke('T',KeyEvent.ALT_MASK));
JComboBox

- Public JComboBox()
- Public JComboBox(Vector items)
- Public JComboBox(ComboBoxModel model)

Constructs a combo box. Can optionally pass a vector or model of items (See default comboBox model for a model implementation)

- Public void addActionListener(ActionListener al)

Causes an action event to be sent to listener al when the user selects or types a new item in the combo box.

JComboBox: Managing Items

- Public void addItem(Object item)
- Public Object getItemAt(int index)
- Public void removeAllItems()
- Public void removeItem(Object item)
- Public void removeItemAt(int index)

JComboBox: Selected Item

- Public int getSelectedIndex()
- Public Object getSelectedItem()
- Public void setSelectedItem(Object item)
- Public void setSelectedIndex(int index)
- Public void setEnabled(boolean enabled)
- Public void setEditable(boolean editable)

If editable, the user can type new arbitrary values into the combo box.

JComboBox Code Example

```java
final JComboBox box = new JComboBox();
box.addItem("Marty"); box.addItem("Tom"); box.addItem("Jessica");
box.addActionListener(new ActionListener() {
```
public void actionPerformed(ActionEvent event) {
    JOptionPane.showMessageDialog(null,
        "You chose " + box.getSelectedItem());
}

getContentPane().add(box, BorderLayout.NORTH);

Lab 09
Java Programming with SWING
GUI Builder-Part III

Button Group
A logical group of radio buttons that ensures that only one is selected at a time
    - public ButtonGroup()
    - public void add(JRadioButton button)

The ButtonGroup is not a graphical component, just a logical group; the RadioButtons
themselves are added to the container, not the ButtonGroup

JTabbedPane
A container that can hold many "tab" subcontainers, each with components in it
    - public JTabbedPane()
    - public JTabbedPane(int tabAlignment)

Constructs a new tabbed pane. Defaults to having the tabs on top; can be set to
TabbedPane.BOTTOM, LEFT, RIGHT, etc.
    - public void addTab(String title, Component comp)
    - public void addTab(String title, Icon icon, Component comp)
    - public void addTab(String title, Icon icon, Component comp, String tooltip)

        Adds the given component as a tab in this tabbed pane. Can optionally use an icon and/or
tool tip.

JTabbedPane methods
    - public void insertTab(String title, Icon icon, Component comp, String tooltip, int index)
public void remove(Component comp)
public void remove(int index)
public void removeAll()
public void setSelectedComponent(Component c)
public void setSelectedIndex(int index)

[JToolBar]
A movable container to hold common buttons, commands, etc
public JToolBar()
public JToolBar(int orientation)
public JToolBar(String title)
public JToolBar(String title, int orientation)
Constructs a new tool bar, optionally with a title and orientation; can be
JToolBar.HORIZONTAL or VERTICAL, defaults to horizontal
public void add(Component comp)
Adds the given component to this tool bar's horizontal/vertical flowing layout.

[JToolBar: Usage]
Construct toolbar
Add items (usually buttons) to toolbar
Add toolbar to edge of borderlayout of content pane (usually north)
Don't put anything in other edges (n/s/e/w)!

[JList]
A list of selectable pre-defined items
public JList()
Constructs an empty JList.
public JList(ListModel model)
public JList(Object[] data)
public JList(Vector data)
Constructs a JList that displays the given data.
public void addListSelectionListener(
ListSelectionListener lsl)
Adds the given listener to be informed when the selected index / indices change for this list.

**JList: more methods**

- public void clearSelection()
- public int getSelectedIndex()
- public int[] getSelectedIndices()
- public Object getSelectedValue()
- public Object[] getSelectedValues()
- public void setSelectedIndex(int index)
- public void setSelectedIndices(int[] indices)

Methods for getting / setting the selected item and index in the list

**JList: even more methods**

- public ListModel getModel()
- public void setListData(Object[] data)
- public void setListData(Vector data)
- public void setModel(ListModel model)

  Causes this list to now display the given collection of data.

- public int getSelectionMode()
- public void setSelectionMode(int mode)

  Get / set selection mode for the list. For example, set to ListSelectionModel.SINGLE_SELECTION to only allow one item in the list to be chosen at once.

**JDialog**

A window that is a child of the overall JFrame; used for popup windows, option/config boxes, etc

- public JDialog(Dialog parent, boolean modal)
- public JDialog(Frame parent, String title, boolean modal)

  Constructs a new dialog with the given parent and title. If the modal flag is set, this dialog is a child of the parent and the parent will be locked until the dialog is closed.

- public void show()
Shows this dialog on screen. If the dialog is modal, calling show() will lock the parent frame/dialog.

- JDialog has most all JFrame methods: getContentPane(), setJMenuBar(JMenuBar), setResizable(boolean), setTitle(String), ...

Problem: positioning, resizing

How does the programmer specify where each component sits in the window, how big each component should be, and what the component should do if the window is resized/moved/maximized/etc?

Absolute positioning (C++, C#, others):
Specify exact pixel coordinates for every component

Layout managers (Java):
Have special objects that decide where to position each component based on some criteria

What are benefits or drawbacks to each approach?

Containers with layout
The idea: Place many components into a special component called a container, then add the container to the JFrame
Container
An object that holds components; it also governs their positions, sizes, and resize behavior

- **Public void add(Component comp)**
  
  **public void add(Component comp, Object info)**

  Adds a component to the container, possibly giving extra information about where to place it.

- **Public void remove(Component comp)**

  Removes the given component from the container.

- **Public void setlayout(LayoutManager mgr)**

  Uses the given layout manager to position the components in the container.

- **Public void validate()**

  You should call this if you change the contents of a container that is already on the screen, to make it re-do its layout.

JPanel
A panel is our container of choice; it is a subclass of Container, so it inherits the methods from the previous slide and defines these additional methods (among others):

- **public JPanel()**

  Constructs a panel with a default flow layout.

- **public JPanel(LayoutManager mgr)**

  Constructs a panel that uses the given layout manager.

Preferred size of components
Swing component objects each have a certain size they would "like" to be--just large enough to fit their contents (text, icons, etc.)

- This is called the preferred size of the component

- Some types of layout managers (e.g. FlowLayout) choose to size the components inside them to the preferred size; others (e.g. BorderLayout, GridLayout) disregard the preferred size and use some other scheme

BorderLayout
- **public BorderLayout()**
– Divides container into five regions: NORTH, SOUTH, WEST, EAST, CENTER
– NORTH and SOUTH regions expand to fill region horizontally, and use preferred size vertically
– WEST and EAST regions expand to fill region vertically, and use preferred size horizontally
– CENTER uses all space not occupied by others

Container panel = new JPanel(new BorderLayout());
panel.add(new JButton("Button 1 (NORTH)", BorderLayout.NORTH);

FlowLayout
Public FlowLayout()
– Treats container as a left-to-right, top-to-bottom "page" or "paragraph"
– Components are given their preferred size both horizontally and vertically
– Components are positioned in order added
– If too long, components wrap around to next line

Container panel = new JPanel(new FlowLayout());
panel.add(new JButton("Button 1"));

GridLayout
Public GridLayout(int rows, int columns)
– Treats container as a grid of equally-sized rows and columns
– Components are given equal horizontal / vertical size, disregarding preferred size
– Can specify 0 rows or columns to indicate expansion in that direction as needed

BoxLayout
Box.createHorizontalBox()
Box.createVerticalBox()
– Aligns components in container in a single row or column
– Components use preferred sizes and align based on their preferred alignment
– Preferred way to construct a container with box layout:
  Box.createHorizontalBox(); or Box.createVerticalBox();

Other layouts
– CardLayout layers of "cards" stacked on top of each other; one visible at a time
– GridBagLayout very complicated; my recommendation: never ever use it
– Custom / null layout allows you to define absolute positions using setX/Y and setWidth/Height

Problem with layout managers

How would you create a complex window like this, using the layout managers shown?

Solution: composite layout

– Create panels within panels
– Each panel has a different layout, and by combining the layouts, more complex / powerful layout can be achieved

Example:

– How many panels?
– What layout in each?
Some eye candy...

Making a Java Swing GUI look like the native operating system:

```java
try {
    UIManager.setLookAndFeel(
        UIManager.getSystemLookAndFeelClassName());
} catch (Exception e) {} \(\)
```

Adding borders to components:

```java
whateverComponent.setBorder(
    BorderFactory.createLineBorder(Color.BLUE, 3));
```

GUI is not done yet...

What is missing?
- Why don't the buttons do anything when we click them?
- How can we fix this problem?

Next topic: events
- (Making the GUI responsive to user interaction)
- Allows our own guis to be interactive like JOptionPane

Event-driven Programming
- Program's execution is indeterminate
- On-screen components cause events to occur when they are clicked / interacted with
- Events can be handled, causing the program to respond, driving the execution thru events
  (an "event-driven" program)
Java Event Hierarchy

```
java.lang.Object
  +--java.util.EventObject
    +--java.awt.AWTEvent
      +--java.awt.event.ActionEvent
      +--java.awt.event.TextEvent
    +--java.awt.event.ComponentEvent
      +--java.awt.event.FocusEvent
      +--java.awt.event.WindowEvent
    +--java.awt.event.InputEvent
      +--java.awt.event.KeyEvent
      +--java.awt.event.MouseEvent
```

import java.awt.event.*;

**Action events (.ActionEvent)**

Most common / simple event type in Swing

Represent an action occurring on a GUI component

Created by:
  – Button clicks
  – Check box checking / unchecking
  – Menu clicks
  – Pressing enter in a text field
  – Etc.

**Listening for events**

  – Attach listener to component
  – Listener’s appropriate method will be called when event occurs (e.g. When the button is clicked)
  – For action events, use actionlistener

**Writing an ActionListener**

```
// part of Java; you don’t write this
public interface ActionListener {
```
public void actionPerformed(ActionEvent event);
}
// Prints a message when the button is clicked.

public class MyActionListener
    implements ActionListener {
    public void actionPerformed(ActionEvent event)
    {
        System.out.println("Event occurred!");
    }
}

**Attaching an ActionListener**

`JButton button = new JButton("button 1");
ActionListener listener = new MyActionListener();
button.addActionListener(listener);`

- Now button will print "Event occurred!" when clicked
- `addActionListener` method exists in many Swing components

**ActionEvent objects**

- Public Object `getSource()`
  - Returns object that caused this event to occur.
- Public String `getActionCommand()`
  - Returns a string that represents this event.
  - (for example, text on button that was clicked)

**Question: Where to put the listener class?**

**Inner class listener**

public class Outer {

    private class Inner implements ActionListener {
        public void actionPerformed(
            ActionEvent event) {

            ...
        }
    }
}
Anonymous inner listener

```java
public class Outer {
    public Outer() {
        JButton myButton = new JButton();
        myButton.addActionListener(new Inner());
    }
}

public class Inner implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ... 
    }
}
```

ActionListener in JFrame

```java
public class Outer extends JFrame implements ActionListener {
    public Outer() {
        JButton myButton = new JButton();
        myButton.addActionListener(this);
    }
    public void actionPerformed(ActionEvent event) {
        ...
    }
}
```
GUI builder software

JBuilder 2005 includes a mature GUI builder

http://www.borland.com/jbuilder/

Reminder: model and view

**Model:** classes in your system that are related to the internal representation of the state of the system

- Often part of the model is connected to file(s) or database(s)
- Examples (card game): card, deck, player
- Examples (bank system): account, user, userlist

**View:** classes in your system that display the state of the model to the user

- Generally, this is your GUI (could also be a text UI)
- Should not contain crucial application data
- Different views can represent the same data in different ways

Example: Bar chart vs. pie chart

Examples: PokerPanel, BankApplet

**Model-view-controller**

**Model-view-controller (MVC):** common design paradigm for graphical systems
Controller: classes that connect model and view
- Defines how user interface reacts to user input (events)
- Receives messages from view (where events come from)
- Sends messages to model (tells what data to display)
- Sometimes part of view (see left)

Lab No. 10
Unit Testing with JUnit

Iterative Software development
**JUnit**

- JUnit is a unit test environment for Java programs developed by Erich Gamma and Kent Beck.
  
  - **Writing test cases**
  - **Executing test cases**
  - **Pass/fail? (expected result = obtained result?)**

- Consists in a framework providing all the tools for testing
  
  - framework: set of classes and conventions to use them.

- It is integrated into eclipse through a graphical plug-in.

**JUnit (3.x and 4.x)**

- Test framework
  
  - Test cases are Java code
  
  - Test case = “sequence of operations + inputs + expected values”
JUnit 3.x for testing programs

- JUnit tests
  - “Substitute the use of `main()` to check the program behaviour”

- All we need to do is:
  - Write a sub-class of `testcase`
  - Add to it one or more `test methods`
  - Run the test using `junit`

Framework elements

**TestCase**

- Base class for classes that contain tests

```
assert()
```

- Method family to *check conditions*

**TestSuite**

- Enables grouping several test cases

```
TestSuite
```

An example

```java
import junit.framework.TestCase;
```
public class StackTester extends TestCase {
    public StackTester(String name) {
        super(name);
    }
    public void testStack() {
        Stack aStack = new Stack();
        if(!aStack.isEmpty()) {
            System.out.println("Stack should be empty!");
            aStack.push(10);
            aStack.push(-4);
            System.out.println("Last element:" + aStack.pop());
            System.out.println("First element: " + aStack.pop());
        }
    }
}

Assert*

They are public methods defined in the base class TestCase
Their names begin with “assert” and are used in test methods

- es. assertTrue(“stack should be empty”, aStack.isEmpty());
If the condition is false:
- Test fails
- Execution skips the rest of the test method
- The message (if any) is printed
If the condition is true:
- Execution continues normally
For a boolean condition
- assertTrue(“message for fail”, condition);
- assertFalse(“message”, condition);
For object, int, long, and byte values
assertEquals(expected_value, expression);
For float and double values

assertEquals(expected, expression, error);
For objects references

assertNull(reference)
- assertNotNull(reference)

**Assert: example**

```java
public void testStack() {
    Stack aStack = new Stack();
    assertTrue("Stack should be empty!", aStack.isEmpty());
    aStack.push(10);
    assertTrue("Stack should not be empty!", !aStack.isEmpty());
    aStack.push(4);
    assertEquals(4, aStack.pop());
    assertEquals(10, aStack.pop());
}
```

**One concept at a time ...**

```java
public class StackTester extends TestCase {
    public void testStackEmpty() {
        Stack aStack = new Stack();
        assertTrue("Stack should be empty!", aStack.isEmpty());
        aStack.push(10);
        assertTrue("Stack should not be empty!", !aStack.isEmpty());
    }

    public void testStackOperations() {
        Stack aStack = new Stack();
        aStack.push(10);
        aStack.push(-4);
        aStack.push(-4);
        assertEquals(-4, aStack.pop());
    }
}
```
assertEquals(10, aStack.pop());
}
}

**Working rule**

For each test case class, JUnit

- Execute all of its public test methods
  - *I.e. Those whose name starts with “test”*
  - Ignores everything else …

Test classes can contain “**helper methods**” provided that are:

- Non public, or
- Whose name does not begin with “test”

**TestSuite**

Groups several test cases:

```java
public class AllTests extends TestSuite {
    public static TestSuite suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(StackTester.class);
        suite.addTestSuite(AnotherTester.class);
        return suite;
    }
}
```

Test of “Exceptions”

There are two cases:

1. We expect a normal behavior and then no exceptions.
2. We expect an anomalous behavior and then an exception.

**We expect a normal behavior …**

```java
try {
```
```
// We call the method with correct parameters
object.method("Parameter");
assertTrue(true); // OK
}

try {
    // we call the method with wrong parameters
    object.method(null);
    fail("method should fail!!");
} catch(PossibleException e){
    assertTrue(true); // OK
}

SetUp() and tearDown()

setUp() method initialize object(s) under test.
    - called before every test method

tearDown() method release object(s) under test
    - called after every test case method.

ShoppingCart cart;
Book book;

protected void setUp() {
    cart = new ShoppingCart();
    book = new Book("JUnit", 29.95);
    cart.addItem(book);
}

Junit in eclipse - Setup

In Eclipse
```
- Create a new project
- Open project’s property window (File -> Properties)
- Select: Java build path
- Select: libraries
- Add Library
- Select Junit
- Select the type 3.x or 4.x

Create a new JUnit test case

- File
  - New
- Junit Test Case
  - Set the parameters:
    - Junit 3.x or 4.x
    - name of the class
    - etc.
Run as JUnit Test

- Run As

- Junit Test
JUnit 3.x and JUnit 4.x
- Most things are about equally easy
  - JUnit 4 can still run JUnit 3 tests
- All the old assertXXX methods are the same
- JUnit 4 has some additional features
- JUnit 4 provides protection against infinite loops
- Junit 4 uses annotations (@)

From JUnit 3.x to 4.x
- JUnit 4 requires Java 5 or newer
- Don’t extend junit.framework.TestCase; just use an ordinary class
- Import org.junit.* and org.junit.Assert.*
  - Use a static import for org.junit.Assert.*
  - Static imports replace inheritance from junit.framework.TestCase

Use annotations instead of special method names:
- Instead of a setUp method, put @Before before some method
- Instead of a tearDown method, put @After before some method

- Instead of beginning test method names with ‘test’, put @Test before each test method

Annotations in J2SE

- J2SE 5 introduces the Metadata feature (data about data)
- Annotations allow you to add decorations to your code (remember javadoc tags: @author)
- Annotations are used for code documentation, compiler processing (@Deprecated), code generation, runtime processing
- New annotations can be created by developers

Annotations in J2SE … an example

- …. @Override — it is a predefined annotation used by the Java compiler
- It informs the compiler that the element (a method) is meant to override an element declared in a superclass
  • // mark method as a superclass method
  • // that has been overridden

@Override

public int overriddenMethod() { …}

- While it's not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with @Override fails in correctly overriding the original method in its superclass, the compiler generates an error.

JUnit 4.x for testing programs

Import the JUnit 4 classes you need

    import org.junit.*;
    import static org.junit.Assert.*;

Declare your (conventional) Java class

    public class MyProgramTest {
Declare any variables you are going to use, e.g., an instance of the class being tested

    MyProgram program;
    int [ ] array;
    int solution;

If needed, define one method to be executed just once, when the class is first loaded. For instance, when we need to connecting to a database

    @BeforeClass
    public static void setUpClass() throws Exception {
        // one-time initialization code
    }

If needed, define one method to be executed just once, to do cleanup after all the tests have been completed

    @AfterClass
    public static void tearDownClass() throws Exception {
        // one-time cleanup code
    }

If needed, define one or more methods to be executed before each test, e.g., typically for initializing values

    @Before
    public void setUp() {
        program = new MyProgram();
        array = new int[] { 1, 2, 3, 4, 5 };
    }

If needed, define one or more methods to be executed after each test, e.g., typically for releasing resources (files, etc.)

    @After
    public void tearDown() {
    }
@Before and @After methods

- More than one @Before and/or @After methods can be defined in a test case

  • Attention: we don’t know in what order they will execute

- We can inherit @Before and @After methods from a superclass; execution is as follows:

  • Execute the @Before methods in the superclass

  • Execute the @Before methods in this class

  • Execute a @Test method in this class

  • Execute the @After methods in this class

  • Execute the @After methods in the superclass

JUnit 4.x for testing programs

- A test method is annotated with @Test

- It takes no parameters, and returns no result.

- All the usual assertXXX methods can be used

@Test
    public void sum() {
        assertEquals(15, program.sum(array));
        assertTrue(program.min(array) > 0);
    }

Additional Features of @Test

To avoid infinite loops, an execution time limit can be used. The time limit is specified in milliseconds. The test fails if the method takes too long.

@Test (timeout=10)
    public void greatBig() {
        assertTrue(program.ackerman(5, 5) > 10e12);
    }

Some method calls should throw an exception. We can specify that an exception is expected. The test will pass if the expected exception is thrown, and fail otherwise.
Parameterized tests

Using `@RunWith(value=Parameterized.class)` and a method `@Parameters`, a test class is executed with several inputs.

```java
@RunWith(value=Parameterized.class)
public class FactorialTest {
    private long expected;
    private int value;
    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[][] { { 1, 0 }, { 1, 1 }, { 2, 2 }, { 120, 5 } });
    }
    public FactorialTest(long expected, int value) { // constructor
        this.expected = expected;
        this.value = value;
    }
    @Test
    public void factorial() {
        assertEquals(expected, new Calculator().factorial(value));
    }
}
```

Test suites

As before, you can define a suite of tests.

```java
@RunWith(value=Suite.class)
@SuiteClasses(value={
    value=test1.class,
    value=test2.class
})
```
public class AllTests {
  ...

Additional features of JUnit 4

- Instead of JUnit 3’s AssertionError, now failed tests throw an AssertionError
- There is now an additional version of assertEquals for arrays of objects:
  assertEquals(Object[], expected, Object[] actual)
- JUnit 3 had an assertEquals(p, p) method for each kind of primitive p, but JUnit 4 only
  has an assertEquals(object, object) and depends on autoboxing

Autoboxing example

- Consider the following method:
  ```java
  "long sum(long x, long y) { return x + y; }
  ```

- and the following test:
  ```java
  @Test
  public void sum() {
    assertEquals(4, s.sum(2, 2));
  }
  ```
  it fails and gives:
  ```
  expected: <4> but was: <4>
  ```

- This is for the autoboxing
  ```
  assertEquals no longer exists for primitives, only for objects!
  ```
  ```
  Hence, the 4 is autoboxed to an Integer, while sum returns a long
  ```
  ```
  The error message means: expected int 4, but got long 4
  ```
  ```
  To make this work, change the 4 to a “4L” or “(long)4”
  ```

When Testing Programs

Test last

The conventional way for testing in which testing follows the implementation
Test first
The Extreme-Programming view in which testing is used as a development tool

Test First Advantages
Each method has associated a testcase
The confidence of our code increases …

It simplifies:
- Refactoring/restructuring
- Maintenance
- The introduction of new functionalities

Test first help to build the documentation
- Testcases are good “use samples”

Programming is more fun …

**Test-first with Junit**

*Junit in practice …*

Existing system: a class current account to manage a bank account

- *Deposit*
- *Withdraw*

Add a new functionality

- *Settlement*
Example:

```java
CurrentAccount cc = new CurrentAccount();
cc.deposit(12);
cc.draw(-8);
cc.deposit(10);
c.settlement()
expected value 14 euro!
```

Add Testcases for the settlement method

**Test First**

class Test_CurrentAccount extends TestCase{
    public void test_settlementVoid() {
        CurrentAccount c = new CurrentAccount();
        assertEquals(0, c.settlement());
    }
    public void test_settlement() {
```
CurrentAccount c = new CurrentAccount();
c.deposit(12);
c.draw(-8);
c.deposit(10);
assertEquals(14, c.settlement());
}

Add the skeleton code of the method

class CurrentAccount {
    int account[];
    int lastMove;
    CurrentAccount()
        lastMove=0;
        account=new int[10];
    }
    public void deposit(int value){

account[lastMove]=value;
lastMove++;
}
public void draw(int value) {
account[lastMove]=value;
lastMove++;
}
public int settlement() {return 0;}
public static void main(String args[]) {}
}
class Test_CurrentAccount extends TestCase{
public void test_settlementVoid() {
currentAccount c = new currentAccount();
assertEquals(0, c.settlement());
}
public void test_settlement() {
currentAccount c = new currentAccount();
c.deposit(12);
c.draw(-8);
c.deposit(10);
assertEquals(14, c.settlement());
}
}
Run Junit (first time)

JUnit

Test class name:

test_currentAccount

Reload classes every run

Runs: 2/2  Errors: 0  Failures: 2

Results:

× test_settlement(test_currentAccount)
× test_settlementAccountVoid(test_currentAccount)

JUnit framework: AssertionError
at test_currentAccount.test_settlement(test_currentAccount.java:18)

Finished: 0.066 seconds
Rework

class CurrentAccount {
    int account[];
    int lastMove;
    CurrentAccount() {
        lastMove=0; account=new int[10];
    }
    public void deposit(int value){ ...}
    public void draw(int value) { ...}
    public int settlement() {
        int result = 0
        for (int i=0; i<account.length; i++) {
            result = result + account[i];
        }
        return result;
    }
    public static void main(String args[]) {"}
class Test_CurrentAccount extends TestCase{
    public void test_settlementVoid() {
        currentAccount c = new currentAccount();
        assertEquals(0, c.settlement());
    }
    public void test_settlement() {
        currentAccount c = new currentAccount();
        c.deposit(12);
        c.draw(-8);
        c.deposit(10);
        assertEquals(14, c.settlement());
    }
}
Run Junit (*second time*)

![JUnit screenshot]

Add a testcase

class CurrentAccount {
    int account[];
    int lastMove;
}
CurrentAccount()
    lastMove=0; account=new int[10];
}
public void deposit(int value){ ...}
public void draw(int value) { ...}
public int settlement() {
    int result = 0
    for (int i=0; i<account.length; i++) {
        result = result + account[i];
    }
    return result;
}
public static void main(String args[]) {}
}
class Test_currentAccount extends TestCase{
...
public void test_realCaseSettlement() {
    currentAccount c = new currentAccount();
    for (int i=0; i < 10; i++)
        c.deposit(1);
    c.draw(-10);
    assertEquals(0, c.settlement());
}  
}
Run JUnit (third time)
Rework

class CurrentAccount {
    int account[];
    int lastMove;
    CurrentAccount(){
        lastMove=0; account=new int[100];
    }
    public void deposit(int value){ ...}
    public void draw(int value) { ...}
    public int settlement() {
        int result = 0
        for (int i=0; i<account.length; i++) {
            result = result + account[i];
        }
        return result;
    }
    public static void main(String args[]){}
}
class Test_currentAccount extends TestCase {
    ...
    public void test_realCaseSettlement() {
        currentAccount c = new currentAccount();
        for (int i=0; i<10; i++)
            c.deposit(1);
        c.draw(-10);
        assertTrue(0, c.settlement());
    }
}
Run JUnit (*fourth time*)

![JUnit screenshot]

Refactoring

```java
public class CurrentAccount {
    List account = new LinkedList();
}
```
public void deposit(int value) {
    account.add(new Integer(value));
}

public void draw(int value) {
    account.add(new Integer(value));
}

public int settlement() {
    int result = 0;
    Iterator it = account.iterator();
    while (it.hasNext()) {
        Integer value_integer = (Integer) it.next();
        int val = value_integer.intValue();
        result = result + val;
    }
    return result;
}
Run JUnit (fifth time)

JUnit

JUnit: it is one of many in the xUnit family....

NUnit
- It is an open source unit testing framework for Microsoft .NET.
- It serves the same purpose as JUnit does in the Java world.

“The tip of the iceberg”

- Coverage testing (JCoverage, Clover, …)
- Integration testing
- System/GUI testing (Jemmy, Abbot, …)
- Testing legacy applications
- Testing J2EE applications
- Testing database applications
- Testing EJBs
- Testing Web applications (HttpUnit, JWebUnit, …)
- Testing Web services
- Testing in isolation with Mock Objects