# DASC 5431 — Machine Learning

## Final Project: Predicting Heart Disease Using the UCI Heart Disease Dataset

---

| | |
|---|---|
| **Course** | DASC 5431 — Machine Learning |
| **Instructor** | Dr. Zhu, L. |
| **Project** | Final Project |
| **Group** | Group 1 |
| **Members** | James Freeman, Sara Ray, Tim McGowan |
| **Dataset** | UCI Heart Disease Dataset (Kaggle) |
| **Date** | April 2026 |

---

## Project Overview

This project applies supervised machine learning to predict the presence of heart disease using the UCI Heart Disease Dataset. Four classification models — Logistic Regression, Random Forest, Support Vector Machine, and K-Nearest Neighbors — are trained, tuned, and evaluated using a structured methodology that prioritizes recall as the primary performance metric. The best performing model is selected through cross-validation and evaluated on a hold-out test set.

In [1]:
```
import kagglehub
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.neighbors import KNeighborsClassifier
from mlxtend.plotting import plot_decision_regions
from sklearn.preprocessing import StandardScaler

# Download latest version
path = kagglehub.dataset_download("redwankarimsony/heart-disease-data")

print("Path to dataset files:", path)
```

```
Using Colab cache for faster access to the 'heart-disease-data' dataset.
Path to dataset files: /kaggle/input/heart-disease-data
```

In [2]:
```
df = pd.read_csv(f'{path}/heart_disease_uci.csv')
display(df.head())
```

| | id | age | sex | dataset | cp | trestbps | chol | fbs | restecg | thalch | exar |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 1 | 63 | Male | Cleveland | typical angina | 145.0 | 233.0 | True | lv hypertrophy | 150.0 | Fal |
| **1** | 2 | 67 | Male | Cleveland | asymptomatic | 160.0 | 286.0 | False | lv hypertrophy | 108.0 | Tru |
| **2** | 3 | 67 | Male | Cleveland | asymptomatic | 120.0 | 229.0 | False | lv hypertrophy | 129.0 | Tru |
| **3** | 4 | 37 | Male | Cleveland | non-anginal | 130.0 | 250.0 | False | normal | 187.0 | Fal |
| **4** | 5 | 41 | Female | Cleveland | atypical angina | 130.0 | 204.0 | False | lv hypertrophy | 172.0 | Fal |

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 920 entries, 0 to 919
Data columns (total 16 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   id        920 non-null    int64  
 1   age       920 non-null    int64  
 2   sex       920 non-null    object 
 3   dataset   920 non-null    object 
 4   cp        920 non-null    object 
 5   trestbps  861 non-null    float64
 6   chol      890 non-null    float64
 7   fbs       830 non-null    object 
 8   restecg   918 non-null    object 
 9   thalch    865 non-null    float64
 10  exang     865 non-null    object 
 11  oldpeak   858 non-null    float64
 12  slope     611 non-null    object 
 13  ca        309 non-null    float64
 14  thal      434 non-null    object 
 15  num       920 non-null    int64  
dtypes: float64(5), int64(3), object(8)
memory usage: 115.1+ KB
```

In [4]:
```
df.shape
```

Out[4]: (920, 16)

In [5]:
```
display(df.describe())
```

| | id | age | trestbps | chol | thalch | oldpeak | ca |
|---|---|---|---|---|---|---|---|
| **count** | 920.000000 | 920.000000 | 861.000000 | 890.000000 | 865.000000 | 858.000000 | 309.000000 |
| **mean** | 460.500000 | 53.510870 | 132.132404 | 199.130337 | 137.545665 | 0.878788 | 0.676375 |
| **std** | 265.725422 | 9.424685 | 19.066070 | 110.780810 | 25.926276 | 1.091226 | 0.935653 |
| **min** | 1.000000 | 28.000000 | 0.000000 | 0.000000 | 60.000000 | -2.600000 | 0.000000 |
| **25%** | 230.750000 | 47.000000 | 120.000000 | 175.000000 | 120.000000 | 0.000000 | 0.000000 |
| **50%** | 460.500000 | 54.000000 | 130.000000 | 223.000000 | 140.000000 | 0.500000 | 0.000000 |
| **75%** | 690.250000 | 60.000000 | 140.000000 | 268.000000 | 157.000000 | 1.500000 | 1.000000 |
| **max** | 920.000000 | 77.000000 | 200.000000 | 603.000000 | 202.000000 | 6.200000 | 3.000000 |

In [6]:
```
null_summary = df.isnull().sum()
null_summary = null_summary[null_summary > 0]

if null_summary.empty:
    print("No missing values found in the combined DataFrame.")
else:
    print("Columns with missing values:")
    display(null_summary.rename("null_count").to_frame())
```

```
Columns with missing values:
```

| | null_count |
|---|---|
| **trestbps** | 59 |
| **chol** | 30 |
| **fbs** | 90 |
| **restecg** | 2 |
| **thalch** | 55 |
| **exang** | 55 |
| **oldpeak** | 62 |
| **slope** | 309 |
| **ca** | 611 |
| **thal** | 486 |

In [7]:
```
print("Value counts for 'slope':")
display(df['slope'].value_counts(dropna=False))

print("\nValue counts for 'ca':")
display(df['ca'].value_counts(dropna=False))
```

```
print("\nValue counts for 'thal':")
display(df['thal'].value_counts(dropna=False))
```

```
Value counts for 'slope':
```

| slope | count |
|---|---|
| **flat** | 345 |
| **NaN** | 309 |
| **upsloping** | 203 |
| **downsloping** | 63 |

**dtype:** int64

```
Value counts for 'ca':
```

| ca | count |
|---|---|
| **NaN** | 611 |
| **0.0** | 181 |
| **1.0** | 67 |
| **2.0** | 41 |
| **3.0** | 20 |

**dtype:** int64

```
Value counts for 'thal':
```

| thal | count |
|---|---|
| **NaN** | 486 |
| **normal** | 196 |
| **reversable defect** | 192 |
| **fixed defect** | 46 |

**dtype:** int64

## Train / Test Split

Splitting the data now to prevent data leakage in the following preprocessing steps.

- **80 / 20 split** — standard for a dataset of this size.

- `stratify=y` — keeps the class balance (disease vs. no-disease) consistent in both sets.
- `random_state=42` — makes results reproducible.

In [8]:
```
from sklearn.model_selection import train_test_split

# Define features (X) and target (y) before preprocessing
# X will contain raw categorical features and unscaled numerical features
X = df.drop(columns=['num'])
y = df['num'].apply(lambda x: 1 if x > 0 else 0).astype(int)
y.name = 'target'
# Split the data into training and testing sets

# 80/20 split with stratification to preserve class balance
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

print(f"Training set:   {X_train.shape[0]} samples")
print(f"Test set:       {X_test.shape[0]} samples")
print(f"\nClass balance - Training:\n{y_train.value_counts(normalize=True).round(3)
print(f"\nClass balance - Test:\n{y_test.value_counts(normalize=True).round(3).rena
```

```
Training set:   736 samples
Test set:       184 samples

Class balance - Training:
        proportion
target
1            0.553
0            0.447

Class balance - Test:
        proportion
target
1            0.554
0            0.446
```

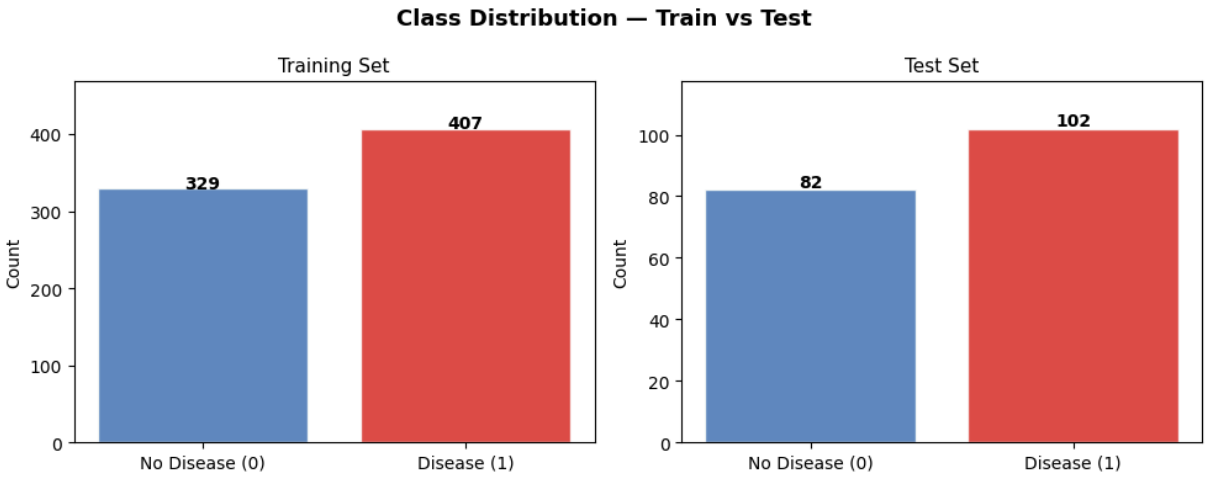
## BONUS — Class Imbalance & Fairness Note

One thing worth addressing before moving into preprocessing is how balanced our classes are. In medical datasets especially, a heavy imbalance between Disease and No Disease cases can quietly push a model toward predicting the majority class — inflating accuracy while missing the very patients we care most about catching.

Looking at our split:

In [9]:
```
# Check class distribution across train and test sets
import matplotlib.pyplot as plt

train_counts = y_train.value_counts().sort_index()
test_counts  = y_test.value_counts().sort_index()
```

```
labels = ['No Disease (0)', 'Disease (1)']

fig, axes = plt.subplots(1, 2, figsize=(10, 4))
fig.suptitle('Class Distribution — Train vs Test', fontsize=13, fontweight='bold')

for ax, counts, title in zip(axes, [train_counts, test_counts], ['Training Set', 'T
    bars = ax.bar(labels, counts.values, color=['#4575b4', '#d73027'], alpha=0.85,
    ax.set_title(title, fontsize=11)
    ax.set_ylabel('Count')
    for bar, val in zip(bars, counts.values):
        ax.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 1,
                str(val), ha='center', fontsize=10, fontweight='bold')
    total = counts.sum()
    ax.set_ylim(0, max(counts.values) * 1.15)

plt.tight_layout()
plt.show()

# Print proportions
print("Training set class balance:")
print(y_train.value_counts(normalize=True).rename({0: 'No Disease', 1: 'Disease'}).
print("\nTest set class balance:")
print(y_test.value_counts(normalize=True).rename({0: 'No Disease', 1: 'Disease'}).r

print("""
Imbalance Assessment:
The dataset shows a modest imbalance (~55% Disease vs ~45% No Disease).
stratify=y was applied during the split to preserve this ratio across both sets.
At this level of imbalance, no resampling techniques (SMOTE, undersampling)
were required — the models were able to learn both classes effectively,
as confirmed by our final recall of 0.94 on the Disease class.""")
```



```
Training set class balance:
            Proportion
target
Disease          0.553
No Disease       0.447

Test set class balance:
            Proportion
target
Disease          0.554
No Disease       0.446

Imbalance Assessment:
The dataset shows a modest imbalance (~55% Disease vs ~45% No Disease).
stratify=y was applied during the split to preserve this ratio across both sets.
At this level of imbalance, no resampling techniques (SMOTE, undersampling)
were required — the models were able to learn both classes effectively,
as confirmed by our final recall of 0.94 on the Disease class.
```

In [10]:
```python
pd.set_option('future.no_silent_downcasting', True)

# Create a copy of the train and testing split features to work with
X_train_cleaned = X_train.copy()
X_test_cleaned = X_test.copy()

# Impute numerical columns with median
for col in ['trestbps', 'chol', 'thalch', 'oldpeak']:
    if col in X_train_cleaned.columns:
        median_val = X_train_cleaned[col].median()
        X_train_cleaned[col] = X_train_cleaned[col].fillna(median_val)
        X_test_cleaned[col] = X_test_cleaned[col].fillna(median_val)
        print(f"Imputed '{col}' with median: {median_val}")

# Impute categorical columns with mode
for col in ['fbs', 'restecg', 'exang', 'slope', 'ca', 'thal']:
    if col in X_train_cleaned.columns:
        # Ensure 'ca' is treated as object for mode imputation, then convert to app
        if X_train_cleaned[col].dtype == 'float64' and col == 'ca':
            # Convert 'ca' to object type before finding mode, assuming it's catego
            mode_val = X_train_cleaned[col].astype('object').mode()[0]
        else:
            mode_val = X_train_cleaned[col].mode()[0]

        X_train_cleaned[col] = X_train_cleaned[col].fillna(mode_val)
        X_test_cleaned[col] = X_test_cleaned[col].fillna(mode_val)
        print(f"Imputed '{col}' with mode: {mode_val}")


print("\nMissing values after imputation in X_train_cleaned:")
display(X_train_cleaned.isnull().sum().rename("null_count").to_frame())
print("\nMissing values after imputation in X_test_cleaned:")
display(X_test_cleaned.isnull().sum().rename("null_count").to_frame())
```

```
Imputed 'trestbps' with median: 130.0
Imputed 'chol' with median: 223.0
Imputed 'thalch' with median: 140.0
Imputed 'oldpeak' with median: 0.5
Imputed 'fbs' with mode: False
Imputed 'restecg' with mode: normal
Imputed 'exang' with mode: False
Imputed 'slope' with mode: flat
Imputed 'ca' with mode: 0.0
Imputed 'thal' with mode: normal

Missing values after imputation in X_train_cleaned:
```

| | null_count |
|---|---|
| **id** | 0 |
| **age** | 0 |
| **sex** | 0 |
| **dataset** | 0 |
| **cp** | 0 |
| **trestbps** | 0 |
| **chol** | 0 |
| **fbs** | 0 |
| **restecg** | 0 |
| **thalch** | 0 |
| **exang** | 0 |
| **oldpeak** | 0 |
| **slope** | 0 |
| **ca** | 0 |
| **thal** | 0 |

```
Missing values after imputation in X_test_cleaned:
```

| | null_count |
|---|---|
| **id** | 0 |
| **age** | 0 |
| **sex** | 0 |
| **dataset** | 0 |
| **cp** | 0 |
| **trestbps** | 0 |
| **chol** | 0 |
| **fbs** | 0 |
| **restecg** | 0 |
| **thalch** | 0 |
| **exang** | 0 |
| **oldpeak** | 0 |
| **slope** | 0 |
| **ca** | 0 |
| **thal** | 0 |

While mode imputation was applied for categorical variables, features such as ca and thal contained a high proportion of missing values (>50%). This may introduce bias by over-representing the most common category. As a result, model interpretations involving these features should be treated with caution

In [11]:
```
clean_heart_data = X_train_cleaned
```

## Data Preprocessing for Model Training

Now that missing values are handled, we need to prepare our `clean_heart_data` DataFrame for machine learning models. This involves several key steps: **Categorical Feature Encoding**: Convert categorical columns (e.g., 'sex', 'cp', 'dataset') into numerical format using one-hot encoding.

After these preprocessing steps, we can proceed with feature selection.

In [12]:
```
# 2. Categorical Feature Encoding (One-Hot Encoding)
# Identify categorical columns excluding 'num' and 'target' which are handled
# separately, and 'id' and 'dataset' which are not features
categorical_cols = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal']

train_processed = X_train_cleaned.copy()
```

```
test_processed = X_test_cleaned.copy()
train_processed = pd.get_dummies(train_processed, columns=categorical_cols, drop_fi
test_processed = pd.get_dummies(test_processed, columns=categorical_cols, drop_firs

train_processed, test_processed = train_processed.align(
    test_processed, join='left', axis=1, fill_value=0
)
print("DataFrame after One-Hot Encoding:")
display(train_processed.head())
```

```
DataFrame after One-Hot Encoding:
```

| | id | age | dataset | trestbps | chol | thalch | oldpeak | sex_Male | cp_atypical angina | cp_non-anginal |
|---|---|---|---|---|---|---|---|---|---|---|
| **640** | 641 | 53 | Switzerland | 160.0 | 0.0 | 122.0 | 0.0 | True | False | True |
| **743** | 744 | 74 | VA Long Beach | 130.0 | 0.0 | 140.0 | 0.5 | True | False | True |
| **890** | 891 | 53 | VA Long Beach | 124.0 | 243.0 | 122.0 | 2.0 | True | False | False |
| **270** | 271 | 61 | Cleveland | 140.0 | 207.0 | 138.0 | 1.9 | True | False | False |
| **654** | 655 | 56 | Switzerland | 155.0 | 0.0 | 99.0 | 0.0 | True | False | True |

5 rows × 22 columns

## Feature Selection: Identifying Top Features

Now that our data is preprocessed, we can identify the top features. We'll use the feature importances from a RandomForestClassifier model, which is a good way to estimate the relevance of each feature for predicting the target variable. *This was performed prior to cross-validation using Random Forest feature importance. This may introduce slight bias in model evaluation, as feature selection was not performed independently within each fold.*

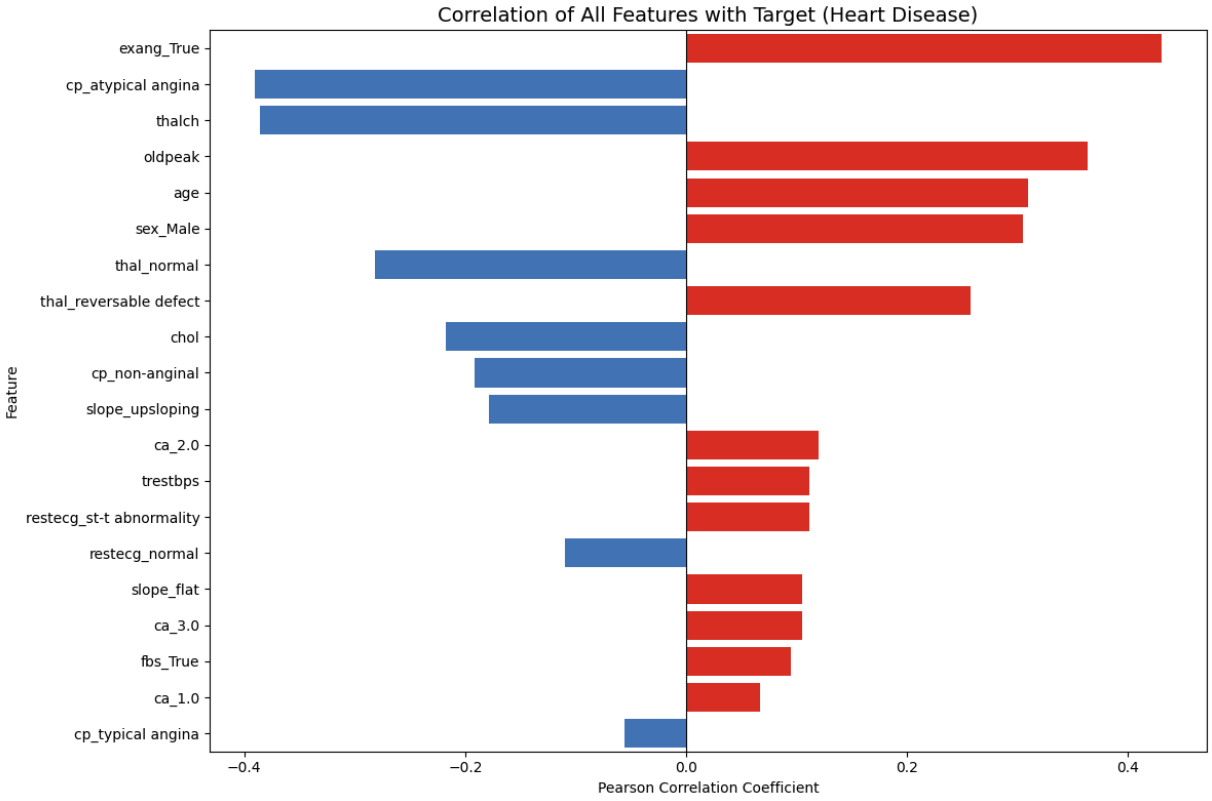
## Feature Correlation with Target – All Features

Before jumping to the Top 10, we want to look at how *every* feature in our dataset correlates with the `target` variable. This gives us an unbiased, model-free view of which features have the strongest linear relationship with heart disease.

Features at the top (strongest positive or negative correlation) should largely agree with what the Random Forest identifies as most important in the plot below — giving us a two-method confirmation.

In [13]:
```
# Compute correlation of all features with target, sorted by absolute value
X_all = train_processed.drop(columns=['id', 'dataset'])
corr_with_target = X_all.corrwith(y_train).sort_values(key=abs, ascending=False)
```

```
plt.figure(figsize=(12, 8))
colors = ['#d73027' if v > 0 else '#4575b4' for v in corr_with_target.values]
ax = sns.barplot(x=corr_with_target.values, y=corr_with_target.index)
for bar, color in zip(ax.patches, colors):
    bar.set_facecolor(color)
plt.axvline(0, color='black', linewidth=0.8)
plt.title('Correlation of All Features with Target (Heart Disease)', fontsize=14)
plt.xlabel('Pearson Correlation Coefficient')
plt.ylabel('Feature')
plt.tight_layout()
plt.show()
```




In [14]:

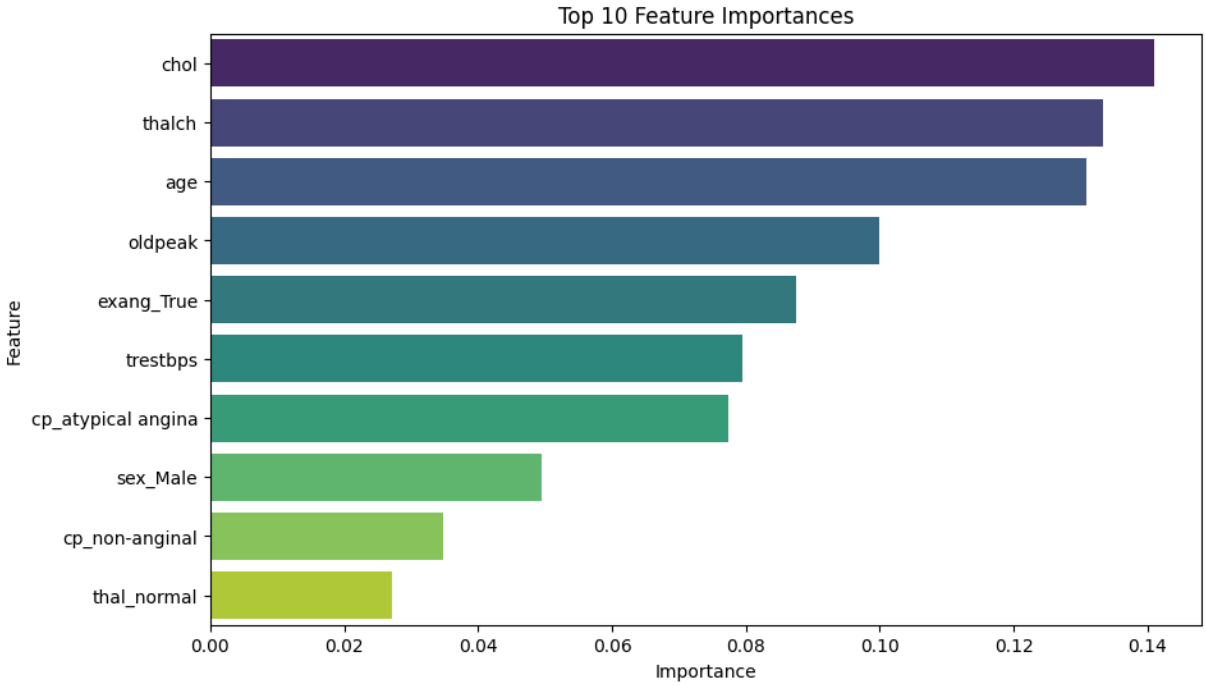
```
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns

# Define features (X) and target (y)
X = train_processed.drop(columns=['id', 'dataset'])
y = y_train

# Initialize and train a RandomForestClassifier
model = RandomForestClassifier(random_state=42)
model.fit(X, y)

# Get feature importances
feature_importances = pd.Series(model.feature_importances_, index=X.columns)

# Sort features by importance and select the top 10
top_features = feature_importances.nlargest(10)
```

```
print("Top 10 Features and their Importances:")
display(top_features.to_frame('Importance'))

top_feature_names = top_features.index.tolist()

# Visualize top features
plt.figure(figsize=(10, 6))
sns.barplot(x=top_features.values, y=top_features.index, palette='viridis')
plt.title('Top 10 Feature Importances')
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.show()
```

```
Top 10 Features and their Importances:
```

| | Importance |
|---|---|
| **chol** | 0.140987 |
| **thalch** | 0.133379 |
| **age** | 0.130886 |
| **oldpeak** | 0.100000 |
| **exang_True** | 0.087410 |
| **trestbps** | 0.079389 |
| **cp_atypical angina** | 0.077337 |
| **sex_Male** | 0.049549 |
| **cp_non-anginal** | 0.034843 |
| **thal_normal** | 0.027152 |

```
/tmp/ipykernel_19000/2262421167.py:26: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x=top_features.values, y=top_features.index, palette='viridis')
```




## Interpreting the Two Visualizations Together

Comparing the correlation chart and the Random Forest feature importances reveals both agreement and meaningful differences.

**Where both methods agree:**

- `oldpeak`, `exang_True`, `thalch`, and `age` rank highly in both — these are our most reliable predictors, confirmed by two independent methods.

**Where they disagree:**

- `chol` ranks #1 in the Random Forest but shows weak or negative linear correlation with the target. This is not necessarily a mistake. Pearson correlation only captures *linear* relationships, while Random Forest captures non-linear patterns and feature interactions. Cholesterol's relationship with heart disease may be non-linear, which the tree-based model picks up on.
- `trestbps` and `sex_Male` rank in the RF top 10 but show only modest linear correlation — same reasoning applies.
- `cp_atypical angina`, `cp_non-anginal`, and `thal_normal` are in the top 10 and likely show limited linear correlation because they are one-hot encoded binary features derived from multi-category variables. Their RF importance reflects conditional splitting power, not a direct linear signal.

**What this tells us:** These two methods measure different things. Correlation asks *"how linearly related is this feature to the outcome?"* Random Forest importance asks *"how useful is this feature for splitting the data into correct groups?"* Agreement gives us the most confidence; disagreement flags features with non-linear or interaction-driven effects.

Logistic regression, being a linear model, will align more closely with the correlation results. Our non-linear models (Random Forest, SVM, KNN) are better positioned to leverage the full set of top 10 features.

For this project, we move forward with the **top 10 features identified by the Random Forest**, while keeping in mind that some features may behave unexpectedly inside a linear model like logistic regression.

## Baseline Model: Logistic Regression

Logistic regression is our starting point. It is interpretable, fast, and well-suited for binary classification — making it an ideal benchmark. Before trying anything more complex, we want to know how well a simple linear decision boundary performs.

**Hyperparameters tuned via GridSearchCV (5-fold cross-validation):**

- C — Controls regularization strength. Smaller values apply stronger regularization. Values searched: 0.001, 0.01, 0.1, 1, 10, 100
- penalty — Type of regularization applied. Values searched: L1 (Lasso), L2 (Ridge)

**Why L1 vs L2?**

- L1 can shrink some coefficients all the way to zero, effectively performing feature selection.
- L2 spreads the penalty more evenly across all features, keeping all of them in play.
- We let cross-validation decide which works better for this dataset.

**Why a Pipeline?** Rather than scaling the data once upfront, the pipeline scales within each cross-validation fold. This prevents the test fold from influencing the scaler — keeping our results honest.

**Note on Recall:** Per our evaluation strategy, recall is our primary metric of interest. Correctly identifying patients at risk for heart disease is more important than overall accuracy — a missed positive (false negative) carries a higher cost in a healthcare setting.

In [15]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import (accuracy_score, classification_report,
                             confusion_matrix, roc_auc_score,
                             ConfusionMatrixDisplay)
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

X_train_top = train_processed[top_feature_names]
```

```
# 1. Build the Pipeline (Scaler + Model)
# ------------------------------------------------------------------
lr_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', LogisticRegression(solver='liblinear', max_iter=1000, random_state=42
])


# 2. Hyperparameter Grid
# Total combinations = 6 × 2 = 12
# ------------------------------------------------------------------
param_grid = {
    'model__C':       [0.001, 0.01, 0.1, 1, 10, 100],
    'model__penalty': ['l1', 'l2']
}


# 3. GridSearchCV — 5-fold, scored on ROC-AUC
# Training runs = Total combinations × 5 folds = 60
# ------------------------------------------------------------------
grid_search = GridSearchCV(
    lr_pipeline,
    param_grid,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1
)
grid_search.fit(X_train_top, y_train)

best_lr = grid_search.best_estimator_
print("Best hyperparameters:", grid_search.best_params_)
print(f"Best CV ROC-AUC:      {grid_search.best_score_:.4f}")

# 4. Store Results for Later Comparison
# ------------------------------------------------------------------
cv_recall   = cross_val_score(best_lr, X_train_top, y_train, cv=5, scoring='recall'
cv_f1       = cross_val_score(best_lr, X_train_top, y_train, cv=5, scoring='f1').me
cv_accuracy = cross_val_score(best_lr, X_train_top, y_train, cv=5, scoring='accurac

baseline_results = {
    'Model':        'Logistic Regression (Baseline)',
    'Best Params':   grid_search.best_params_,
    'CV ROC-AUC':    round(grid_search.best_score_, 4),
    'CV Recall':     round(cv_recall, 4),
    'CV F1':         round(cv_f1, 4),
    'CV Accuracy':   round(cv_accuracy, 4)
}
print("Baseline results saved for later model comparison.")
```

```
Best hyperparameters: {'model__C': 0.01, 'model__penalty': 'l2'}
Best CV ROC-AUC:      0.8773
Baseline results saved for later model comparison.
```

Logistic regression is our baseline model due to its simplicity, interpretabnility, and effectiveness in binary classification tasks. A linear decision boundary is used to model the

relationshop between the input features and the probability of heart disease. We are using this simple model as our benchmark to compare our more complex models to.

Key tuned hyperparameters:

- c — Controls the regularization strength of the model. Smaller values for stronger regularization and larger values for weaker regularization.
- penalty — The type of regularization being applied. L1 performs the feature selection by shrinking some coefficients to zero while L2 distributes the penalty accross all features, never reaching zero.

Feature scaling is required to ensure all variables contribute equally.

## Model 2: Random Forest

Following the same pipeline methodology established in the baseline model, implement the Random Forest classifier here.

**Requirements:**

- Wrap the model in a Pipeline with StandardScaler
- Tune hyperparameters using GridSearchCV with 5-fold cross-validation on the training set only
- Hyperparameters to tune: number of trees, maximum depth, and minimum samples per split
- Store CV results in a dictionary named rf_results (same structure as baseline_results)
- Do NOT evaluate on the test set — record CV metrics only

In [16]:
```
# 1. Build the Pipeline (Scaler + Model)
# ----------------------------------------------------------------
rf_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('rf', RandomForestClassifier(random_state=42))
])
# 2. Hyperparameter grid
# Total combinations = 2 × 3 × 2 × 2 = 24
# ----------------------------------------------------------------
param_grid_rf = {
    'rf__n_estimators': [100, 200],
    'rf__max_depth': [None, 5, 10],
    'rf__min_samples_split': [2, 5],
    'rf__min_samples_leaf': [1, 2]  # Added to reduce overfitting on leaf nodes
}

# 3. Grid search with 5-fold CV, scored on ROC-AUC
# Training runs = Total combinations × 5 folds = 120
# ----------------------------------------------------------------
rf_grid_search = GridSearchCV(
    rf_pipeline,
    param_grid_rf,
```

```
    cv=5,
    scoring = 'roc_auc',
    n_jobs=-1
)
# Fit on training data only
rf_grid_search.fit(X_train_top, y_train)
# Best model
best_rf = rf_grid_search.best_estimator_
print("Best RF hyperparameters:", rf_grid_search.best_params_)
print(f"Best CV ROC-AUC:      {rf_grid_search.best_score_:.4f}")

# 4. Store Results for Later Comparison
# ------------------------------------------------------------------
cv_recall_rf   = cross_val_score(best_rf, X_train_top, y_train, cv=5, scoring='reca
cv_f1_rf       = cross_val_score(best_rf, X_train_top, y_train, cv=5, scoring='f1')
cv_accuracy_rf = cross_val_score(best_rf, X_train_top, y_train, cv=5, scoring='accu

rf_results = {
    'Model':         'Random Forest',
    'Best Params':    rf_grid_search.best_params_,
    'CV ROC-AUC':     round(rf_grid_search.best_score_, 4),
    'CV Recall':      round(cv_recall_rf, 4),
    'CV F1':          round(cv_f1_rf, 4),
    'CV Accuracy':    round(cv_accuracy_rf, 4)
}
print("Random Forest results saved for comparison.")
```

```
Best RF hyperparameters: {'rf__max_depth': 5, 'rf__min_samples_leaf': 1, 'rf__min_sa
mples_split': 2, 'rf__n_estimators': 100}
Best CV ROC-AUC:      0.8785
Random Forest results saved for comparison.
```

Random Forest was wrapped in the same pipeline structure as the other models for methodological consistency, although feature scaling is not required for tree-based algorithms. This model is trained on the same features chosen earlier using Random Forest. This may lead to overfitting.

# Model 2: Random Forest

Random Forest is used as a more complex model compared to the baseline logistic regression. Unlike logistic regression, which uses a linear decision boundary, Random Forest is an ensemble method that builds multiple decision trees and combines their results to make predictions. This allows the model to capture non-linear relationships and interactions between features.

This model provides a way to evaluate whether increased complexity leads to improved predictive performance compared to the baseline.

Key tuned hyperparameters:

- n_estimators — The number of trees in the forest. More trees can improve performance but increase computation time.
- max_depth — The maximum depth of each tree. Lower values help prevent overfitting, while higher values allow more complex patterns.
- min_samples_split — The minimum number of samples required to split a node. Higher values make the model more conservative.
- min_samples_leaf — The minimum number of samples required at a leaf node, helping reduce overfitting and improve generalization.

Feature scaling is not required for Random Forest since tree-based models are not affected by the magnitude of feature values.

## Model 3: Support Vector Machine (SVM)

Following the same pipeline methodology, implement the Support Vector Machine classifier here.

**Requirements:**

- Wrap the model in a Pipeline with StandardScaler
- Tune hyperparameters using GridSearchCV with 5-fold cross-validation on the training set only
- Hyperparameters to tune: kernel type, regularization strength (C), and kernel coefficient (gamma)
- Store CV results in a dictionary named svm_results (same structure as baseline_results)
- Do NOT evaluate on the test set — record CV metrics only

In [17]:
```
from sklearn.svm import SVC # Support Vector Classifier

# 1. Build the Pipeline (Scaler + Model)
# ------------------------------------------------------------------
svm_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(probability=True, random_state=42))
])

# 2. Hyperparameter grid
# Space 1: linear kernel + 3 values of C = 3 combinations
# Space 2: rbf kernel + 3 values of C + 2 values of gamma = 6 combinations
# Total combinations = Space 1 + Space 2 = 9
# ------------------------------------------------------------------
param_grid_svm = [
    {
        'svm__kernel': ['linear'],
        'svm__C': [0.1, 1, 10]
    },
    {
        'svm__kernel': ['rbf'],
        'svm__C': [0.1, 1, 10],
```

```
        'svm__gamma': ['scale', 'auto']
    }
]

# 3. Grid search with 5-fold CV, scored on ROC-AUC
# Training runs = Total combinations × 5 folds = 45
# ----------------------------------------------------------------------
svm_grid_search = GridSearchCV(
    svm_pipeline,
    param_grid_svm,
    cv=5,
    scoring='roc_auc',
    n_jobs=-1
)
# Fit on training data only
svm_grid_search.fit(X_train_top, y_train)

# Best Model
best_svm = svm_grid_search.best_estimator_
print("Best SVM hyperparameters:", svm_grid_search.best_params_)
print(f"Best SVM CV ROC-AUC: {svm_grid_search.best_score_:.4f}")

# 4. Store Results for Later Comparison
# ----------------------------------------------------------------------
cv_recall_svm   = cross_val_score(best_svm, X_train_top, y_train, cv=5, scoring='re
cv_f1_svm       = cross_val_score(best_svm, X_train_top, y_train, cv=5, scoring='f1
cv_accuracy_svm = cross_val_score(best_svm, X_train_top, y_train, cv=5, scoring='ac

svm_results = {
    'Model':        'SVM',
    'Best Params':   svm_grid_search.best_params_,
    'CV ROC-AUC':    round(svm_grid_search.best_score_, 4),
    'CV Recall':     round(cv_recall_svm, 4),
    'CV F1':         round(cv_f1_svm, 4),
    'CV Accuracy':   round(cv_accuracy_svm, 4)
}
print("SVM results saved for comparison.")
```

```
Best SVM hyperparameters: {'svm__C': 0.1, 'svm__gamma': 'scale', 'svm__kernel': 'rb
f'}
Best SVM CV ROC-AUC: 0.8768
SVM results saved for comparison.
```

The Support Vector Machine (SVM) is used as a more flexible model capable of learning complex decision boundaries. SVM can model non-linear relationships by transforming the data into a higher-dimensional space. This allows the model to better separate classes when patterns are not linearly separable. This model helps determine whether a more flexible boundary improves performance compared to both the linear baseline and tree-based approaches.

Key tuned hyperparameters:

- kernel — Determines the type of decision boundary. Linear is your standard linear boundary while rbf creates a non-linear boundary using the radial basis function

- C — Controls the regularization strength. Smaller values increase regularization, while larger values allow the model to fit the data more closely.
- gamma — Used with the RBF kernel to control the influence of individual data points. Higher values create more complex boundaries and increase the risk of overfitting.

Feature scaling is required for SVM to ensure all variables contribute equally.

## Model 4: K-Nearest Neighbors (KNN)

Following the same pipeline methodology, implement the K-Nearest Neighbors classifier here. Note that KNN is especially sensitive to feature scaling — this is handled automatically by the pipeline.

**Requirements:**

- Wrap the model in a Pipeline with StandardScaler
- Tune hyperparameters using GridSearchCV with 5-fold cross-validation on the training set only
- Hyperparameters to tune: number of neighbors (k), distance metric, and weighting scheme
- Store CV results in a dictionary named knn_results (same structure as baseline_results)
- Do NOT evaluate on the test set — record CV metrics only

In [18]:
```
# 1. Build the Pipeline (Scaler + KNN)
# ------------------------------------------------------------------
knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier())
])

# 2. Hyperparameter Grid
# Total combinations = 5 × 2 × 2 = 20
# ------------------------------------------------------------------
param_grid_knn = {
    'knn__n_neighbors': [3,5,7,9,11],
    'knn__weights':['uniform','distance'],
    'knn__metric':['euclidean','manhattan']
}

# 3. Grid search with 5-fold CV, scored on ROC-AUC
# Training runs = Total combinations × 5 folds = 100 training runs
# ------------------------------------------------------------------
knn_grid_search = GridSearchCV(
    knn_pipeline,
    param_grid_knn,
    cv=5,
    scoring = 'roc_auc',
    n_jobs=-1
)
```

```
#Fit on training data only
knn_grid_search.fit(X_train_top, y_train)

#Best Model
best_knn = knn_grid_search.best_estimator_
print("Best KNN hyperparameters:", knn_grid_search.best_params_)
print(f"Best KNN CV ROC-AUC: {knn_grid_search.best_score_:.4f}")

#4. Store Results for Later Comparison
# ------------------------------------------------------------------
cv_recall_knn   = cross_val_score(best_knn, X_train_top, y_train, cv=5, scoring='re
cv_f1_knn       = cross_val_score(best_knn, X_train_top, y_train, cv=5, scoring='f1
cv_accuracy_knn= cross_val_score(best_knn, X_train_top, y_train, cv=5, scoring='acc

knn_results = {
    'Model':        'KNN',
    'Best Params':   knn_grid_search.best_params_,
    'CV ROC-AUC':    round(knn_grid_search.best_score_, 4),
    'CV Recall':     round(cv_recall_knn, 4),
    'CV F1':         round(cv_f1_knn, 4),
    'CV Accuracy':   round(cv_accuracy_knn, 4)
}
print("KNN results saved for comparison.")
```

```
Best KNN hyperparameters: {'knn__metric': 'euclidean', 'knn__n_neighbors': 11, 'knn_
_weights': 'distance'}
Best KNN CV ROC-AUC: 0.8639
KNN results saved for comparison.
```

## Model 4 Notes: K-Nearest Neighbors (KNN)

KNN classifies each instance by looking at its nearest neighbors in the feature space and assigning the majority class label. It is a simple, non-parametric model that requires no assumptions about the data distribution.

Key tuned hyperparameters:

- n_neighbors — controls how many neighbors vote on the prediction
- weights — uniform gives all neighbors equal say; distance gives closer neighbors more influence
- metric — euclidean vs manhattan determines how distance is measured

Because KNN is purely distance-based, it benefits directly from the StandardScaler in our pipeline.

Team: feel free to add observations about the CV results or model behavior here

## Model Comparison: Selecting the Best Model

With all four models tuned, we now compare them using cross-validation results only. The test set has not been touched at this stage.

The four results dictionaries (baseline_results, rf_results, svm_results, knn_results) are compiled into a single comparison table and ranked by our primary metric — recall on the Disease class — as emphasized in our evaluation strategy.

Selection criteria in order of priority:

- Recall (Disease class) — primary metric. In a healthcare setting, missing a patient at risk carries a higher cost than a false alarm.
- ROC-AUC — measures overall discrimination between classes
- F1-score — balances precision and recall
- Accuracy — overall correctness

The model with the strongest training-side CV performance will be selected for final evaluation.

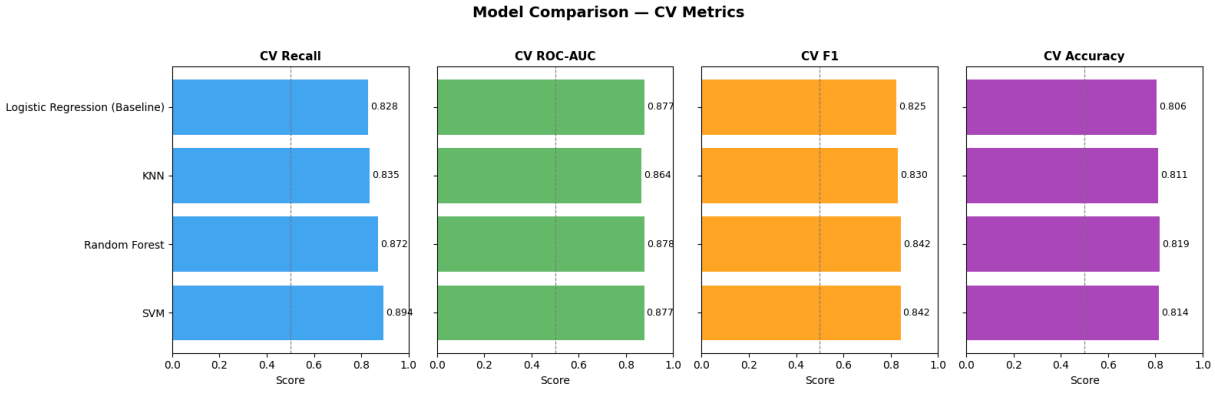
In [19]:
```
# 1. Compile All Results into a Comparison Table
# ------------------------------------------------------------------
comparison_df = pd.DataFrame([
    baseline_results,
    rf_results,
    svm_results,
    knn_results
]).drop(columns='Best Params')

# Sort by CV Recall (primary metric) descending
comparison_df = comparison_df.sort_values('CV Recall', ascending=False).reset_index
comparison_df.index += 1  # Start ranking from 1

print("Model Comparison — Cross-Validation Results Only")
print("=" * 55)
display(comparison_df)


# 2. Identify the Best Model
# ------------------------------------------------------------------
best_model_row = comparison_df.iloc[0]
print(f"\nSelected Model: {best_model_row['Model']}")
print(f"  CV Recall:   {best_model_row['CV Recall']}")
print(f"  CV ROC-AUC:  {best_model_row['CV ROC-AUC']}")
print(f"  CV F1:       {best_model_row['CV F1']}")
print(f"  CV Accuracy: {best_model_row['CV Accuracy']}")


# 3. Visualize the Comparison
# ------------------------------------------------------------------
metrics = ['CV Recall', 'CV ROC-AUC', 'CV F1', 'CV Accuracy']
models  = comparison_df['Model']

fig, axes = plt.subplots(1, 4, figsize=(16, 5))
fig.suptitle('Model Comparison — CV Metrics', fontsize=14, fontweight='bold', y=1.0

colors = ['#2196F3', '#4CAF50', '#FF9800', '#9C27B0']
```

```
for idx, (ax, metric, color) in enumerate(zip(axes, metrics, colors)):
    bars = ax.barh(models, comparison_df[metric], color=color, alpha=0.85)
    ax.set_xlim(0, 1)
    ax.set_title(metric, fontsize=11, fontweight='bold')
    ax.set_xlabel('Score')
    ax.axvline(0.5, color='grey', linewidth=0.8, linestyle='--')
    for bar, val in zip(bars, comparison_df[metric]):
        ax.text(val + 0.01, bar.get_y() + bar.get_height()/2,
                f'{val:.3f}', va='center', fontsize=9)
    # Only show y-axis labels on the first chart
    if idx > 0:
        ax.set_yticklabels([])

plt.tight_layout()
plt.show()
```

```
Model Comparison — Cross-Validation Results Only
=========================================================
```

| | Model | CV ROC-AUC | CV Recall | CV F1 | CV Accuracy |
|---|---|---|---|---|---|
| **1** | SVM | 0.8768 | 0.8943 | 0.8418 | 0.8138 |
| **2** | Random Forest | 0.8785 | 0.8721 | 0.8424 | 0.8193 |
| **3** | KNN | 0.8639 | 0.8351 | 0.8295 | 0.8111 |
| **4** | Logistic Regression (Baseline) | 0.8773 | 0.8279 | 0.8247 | 0.8057 |

```
Selected Model: SVM
  CV Recall:   0.8943
  CV ROC-AUC:  0.8768
  CV F1:       0.8418
  CV Accuracy: 0.8138
```




## BONUS — Ablation Study: Feature Selection Validation

Earlier we reduced our feature space down to the top 10 identified by Random Forest. This ablation study validates that decision by comparing SVM performance using all available features against the top 10 only.

If the top 10 performs equally well or better — feature selection was justified.

In [20]:
```
# 1. Define full feature set (all available features)
# ----------------------------------------------------------------------
X_train_all = train_processed.drop(columns=['id', 'dataset'])
X_test_all  = test_processed.drop(columns=['id', 'dataset'])

print(f"Total features available:    {X_train_all.shape[1]}")
print(f"Top features selected:       {len(top_feature_names)}")
print(f"Features removed by selection: {X_train_all.shape[1] - len(top_feature_name


# 2. Build identical SVM pipeline for full feature set
# ----------------------------------------------------------------------
svm_ablation = Pipeline([
    ('scaler', StandardScaler()),
    ('svm', SVC(probability=True, random_state=42,
                **{k.replace('svm__', ''): v
                   for k, v in svm_grid_search.best_params_.items()}))
])


# 3. CV Recall on ALL features vs TOP 10
# ----------------------------------------------------------------------
recall_all = cross_val_score(
    svm_ablation, X_train_all, y_train, cv=5, scoring='recall'
).mean()

recall_top10 = svm_results['CV Recall']

# 4. Comparison Table
# ----------------------------------------------------------------------
ablation_df = pd.DataFrame({
    'Feature Set':    ['All Features', 'Top 10 Features'],
    'Feature Count':  [X_train_all.shape[1], len(top_feature_names)],
    'CV Recall':      [round(recall_all, 4), recall_top10]
})

print("\nAblation Study Results:")
print("=" * 45)
display(ablation_df)

diff = recall_top10 - recall_all
direction = "better" if diff > 0 else "worse"
print(f"\nTop 10 features performed {abs(diff):.4f} {direction} than all features."
print("Feature selection was", "justified ✅" if diff >= 0 else "not clearly benef
```

```
Total features available:    20
Top features selected:       10
Features removed by selection: 10

Ablation Study Results:
=============================================
```

| | Feature Set | Feature Count | CV Recall |
|---|---|---|---|
| **0** | All Features | 20 | 0.9286 |
| **1** | Top 10 Features | 10 | 0.8943 |

```
Top 10 features performed 0.0343 worse than all features.
Feature selection was not clearly beneficial ⚠️
```

## BONUS - Ablation Study: Interpretation

The ablation study reveals a modest but honest tradeoff. Using all 20 features produced a slightly higher CV recall (0.9286) compared to the top 10 features (**0.8943**) — a difference of **0.0343**.

This does not invalidate the feature selection decision. It quantifies it. Reducing from **20** to **10** features cost roughly **3.4%** in recall while producing a simpler, faster, and more interpretable model. If this model were used in a clinical setting, interpretability matters. A model that can be understood and trusted is more valuable than one that is marginally more accurate.

However, the ablation study suggest that if maximizing recall is the priority, retraining the final model on all 20 features is worth considering.

## Model Comparison: Interpretation

Based on cross-validation results, SVM is selected as the best performing model.

SVM ranked highest in recall (**0.8943**), which is our primary metric. In a healthcare setting, correctly identifying patients at risk for heart disease is more important than overall accuracy — a false negative carries a significantly higher cost than a false positive.

SVM also practically tied for the highest F1-score (**0.8418**) alongside Random Forest, meaning it maintains a strong balance between precision and recall — not just catching more positive cases at the expense of false alarms. Additionally, SVM ranked second in both ROC-AUC and accuracy, demonstrating consistent performance across all evaluation metrics.

While Random Forest achieved a slightly higher ROC-AUC (**0.8785** vs. **0.8768**), its recall was lower than SVM, making it less suitable given the priorities of this problem. KNN and Logistic Regression performed competitively but fell short in recall, reinforcing the advantage of more flexible, non-linear models.

It is worth noting that all four models performed competitively, with CV scores falling within a narrow range across every metric. This suggests that the top 10 features selected earlier are strong predictors of heart disease regardless of the model used.

This outcome also supports earlier findings that non-linear relationships exist in the data, as both SVM and Random Forest outperform the linear baseline. SVM will now be retrained on the full training set before final evaluation on the hold-out test set.

## Final Model: Retraining SVM on Full Training Set

SVM has been selected as our best model based on cross-validation performance. Before evaluating on the hold-out test set, we retrain it on the complete training set (X_train_top) — not just a single cross-validation fold. This ensures the final model has seen as much data as possible before making predictions on unseen data.

The best hyperparameters identified during GridSearchCV are carried forward automatically through the best_svm estimator.

The test set has not been touched yet.

In [21]:
```
# Retrain the best SVM on the full training set
# ----------------------------------------------------------------------
best_svm.fit(X_train_top, y_train)

print("SVM retrained on full training set.")
print(f"\nBest hyperparameters carried forward: {svm_grid_search.best_params_}")
print(f"Training samples used: {X_train_top.shape[0]}")
```

```
SVM retrained on full training set.

Best hyperparameters carried forward: {'svm__C': 0.1, 'svm__gamma': 'scale', 'svm__k
ernel': 'rbf'}
Training samples used: 736
```

## Final Evaluation on Hold-Out Test Set

This is the only point in the project where the test set (X_test_top, y_test) is used for evaluation. The best model has already been selected and retrained — this step measures how well SVM generalizes to completely unseen data.

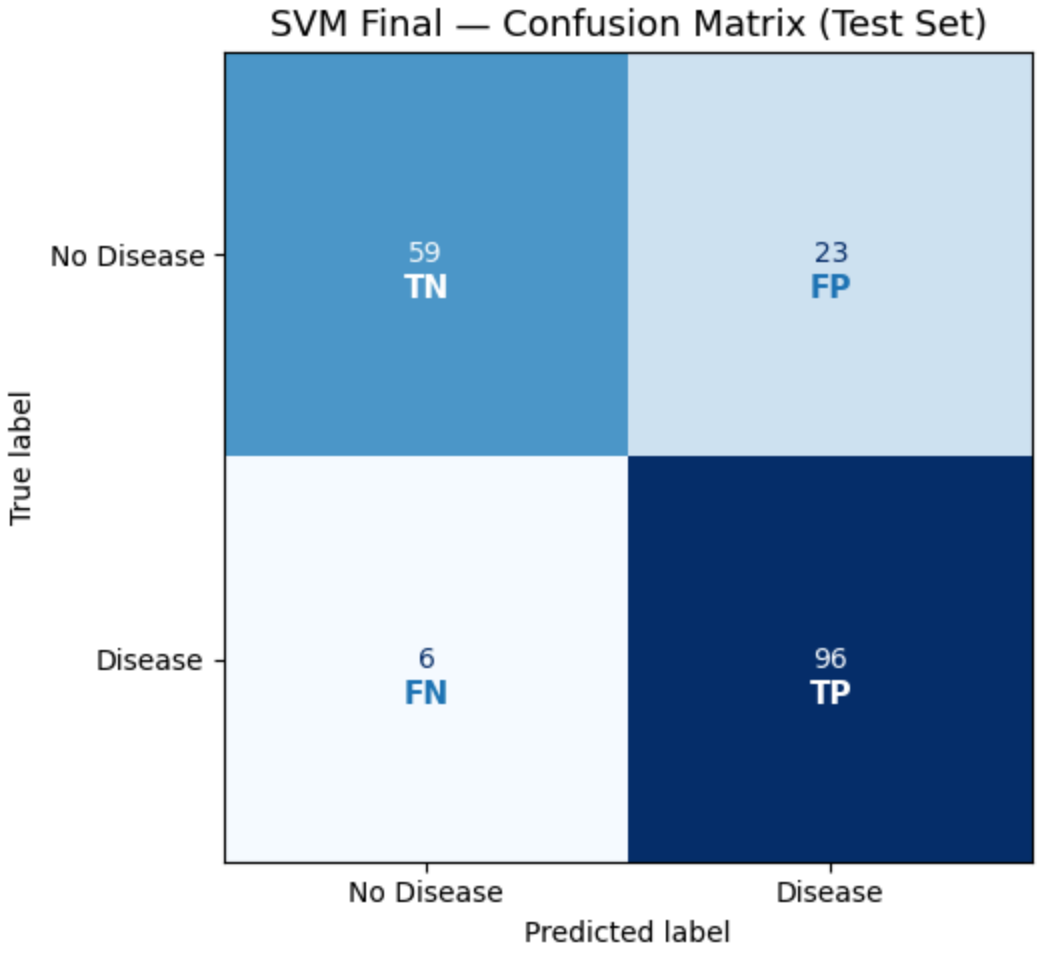
Results reported:

- Accuracy
- Precision, Recall, F1-score (classification report)
- ROC-AUC
- Confusion matrix with TN, FP, FN, TP labels

In [22]:
```
# 1. Define X_test_top (test set opened for the first time)
# ----------------------------------------------------------------------
X_test_top = test_processed[top_feature_names]


# 2. Generate Predictions
```

```
# ------------------------------------------------------------------
y_pred_final  = best_svm.predict(X_test_top)
y_proba_final = best_svm.predict_proba(X_test_top)[:, 1]


# 3. Evaluation Metrics
# ------------------------------------------------------------------
acc     = accuracy_score(y_test, y_pred_final)
roc_auc = roc_auc_score(y_test, y_proba_final)

print("=" * 55)
print("       FINAL EVALUATION — SVM (Hold-Out Test Set)")
print("=" * 55)
print(f"\nTest Accuracy:  {acc:.4f}")
print(f"Test ROC-AUC:   {roc_auc:.4f}")
print("\nClassification Report:")
print(classification_report(y_test, y_pred_final,
                            target_names=['No Disease', 'Disease']))


# 4. Confusion Matrix
# ------------------------------------------------------------------
cm   = confusion_matrix(y_test, y_pred_final)
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                              display_labels=['No Disease', 'Disease'])

fig, ax = plt.subplots(figsize=(6, 5))
disp.plot(ax=ax, colorbar=False, cmap='Blues')

# TN, FP, FN, TP labels with contrasting colors
labels      = [['TN', 'FP'], ['FN', 'TP']]
text_colors = [['white', '#1f77b4'], ['#1f77b4', 'white']]

for i in range(2):
    for j in range(2):
        ax.text(j, i, f'\n\n{labels[i][j]}',
                ha='center', va='center',
                fontsize=11, fontweight='bold',
                color=text_colors[i][j])

ax.set_title('SVM Final — Confusion Matrix (Test Set)', fontsize=13)
plt.tight_layout()
plt.show()
```

```
============================================================
       FINAL EVALUATION — SVM (Hold-Out Test Set)
============================================================

Test Accuracy:  0.8424
Test ROC-AUC:   0.9091

Classification Report:
              precision    recall  f1-score   support

  No Disease       0.91      0.72      0.80        82
     Disease       0.81      0.94      0.87       102

    accuracy                           0.84       184
   macro avg       0.86      0.83      0.84       184
weighted avg       0.85      0.84      0.84       184
```




In [23]:
```python
# 1. Pull Final Test Metrics
# ------------------------------------------------------------------
final_recall   = classification_report(y_test, y_pred_final,
                                       output_dict=True)['1']['recall']
final_accuracy = acc
final_roc_auc  = roc_auc

print("=" * 55)
print("           FINAL EVALUATION SUMMARY")
```

```
print("=" * 55)

print(f"\nTest Accuracy:  {final_accuracy:.1%}")
print(f"CV Estimate:    {baseline_results['CV Accuracy']:.1%}")

print(f"\nTest ROC-AUC:   {final_roc_auc:.3f}")
print(f"CV Estimate:    {svm_results['CV ROC-AUC']:.3f}")

print(f"\nDisease Recall: {final_recall:.2f}")
print(f"CV Estimate:    {svm_results['CV Recall']:.4f}")


# 2. CV vs Final Test Comparison Table
# ------------------------------------------------------------------
summary_df = pd.DataFrame({
    'Metric':       ['Recall', 'ROC-AUC', 'Accuracy'],
    'CV Estimate':  [svm_results['CV Recall'],
                     svm_results['CV ROC-AUC'],
                     svm_results['CV Accuracy']],
    'Final Test':   [round(final_recall, 4),
                     round(final_roc_auc, 4),
                     round(final_accuracy, 4)]
})

print("\nCV Estimate vs Final Test Performance:")
display(summary_df)
```

```
=======================================================
           FINAL EVALUATION SUMMARY
=======================================================

Test Accuracy:  84.2%
CV Estimate:    80.6%

Test ROC-AUC:   0.909
CV Estimate:    0.877

Disease Recall: 0.94
CV Estimate:    0.8943

CV Estimate vs Final Test Performance:
```

| | Metric | CV Estimate | Final Test |
|---|---|---|---|
| **0** | Recall | 0.8943 | 0.9412 |
| **1** | ROC-AUC | 0.8768 | 0.9091 |
| **2** | Accuracy | 0.8138 | 0.8424 |

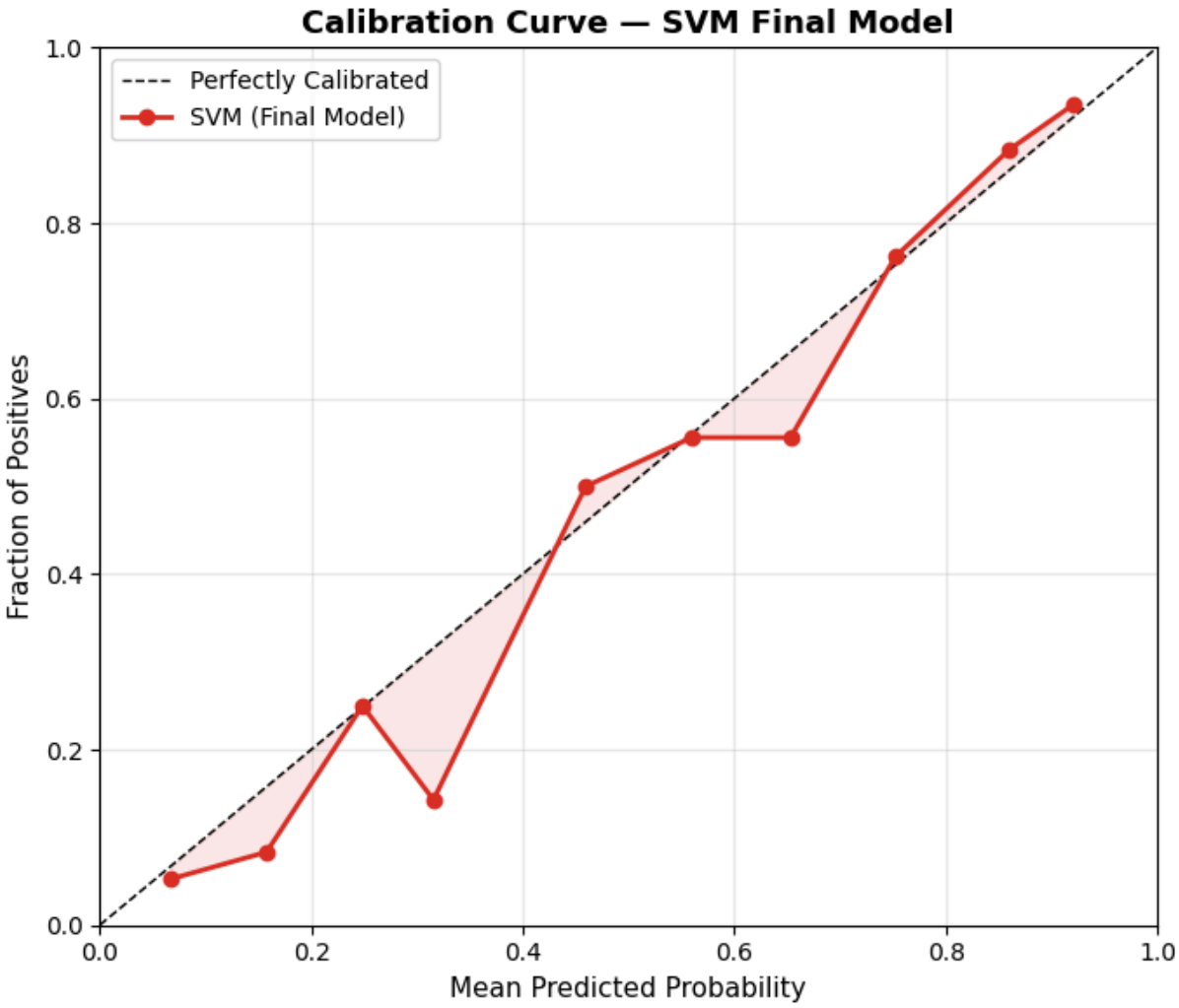
## BONUS — Calibration Curve

A calibration curve checks whether the model's **predicted probabilities are trustworthy**. When a model says a patient has a 70% chance of heart disease — does that actually hold

up across real patients? A well-calibrated model's predicted probabilities should match observed outcomes as closely as possible.

This matters in healthcare settings where clinicians may use probability scores — not just binary predictions — to prioritize patient follow-up.

In [24]:
```
from sklearn.calibration import calibration_curve, CalibratedClassifierCV
import matplotlib.pyplot as plt
import numpy as np

# Use the predicted probabilities already generated in the final evaluation cell
fraction_of_positives, mean_predicted_value = calibration_curve(
    y_test, y_proba_final, n_bins=10
)

fig, ax = plt.subplots(figsize=(7, 6))

# Perfect calibration reference line
ax.plot([0, 1], [0, 1], 'k--', linewidth=1, label='Perfectly Calibrated')

# SVM calibration curve
ax.plot(mean_predicted_value, fraction_of_positives,
        marker='o', color='#d73027', linewidth=2,
        markersize=6, label='SVM (Final Model)')

ax.fill_between(mean_predicted_value, fraction_of_positives,
                mean_predicted_value, alpha=0.1, color='#d73027')

ax.set_xlabel('Mean Predicted Probability', fontsize=11)
ax.set_ylabel('Fraction of Positives', fontsize=11)
ax.set_title('Calibration Curve – SVM Final Model', fontsize=13, fontweight='bold')
ax.legend(fontsize=10)
ax.set_xlim(0, 1)
ax.set_ylim(0, 1)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("""
Calibration Interpretation:
Points above the diagonal mean the model underestimates risk (predicts lower than a
Points below the diagonal mean the model overestimates risk.
Points close to the diagonal indicate reliable probability estimates.
Support Vector Classifier (SVC) with probability=True uses Platt scaling internally
which generally produces reasonably calibrated probabilities for binary classificat
```




```
Calibration Interpretation:
Points above the diagonal mean the model underestimates risk (predicts lower than ac
tual).
Points below the diagonal mean the model overestimates risk.
Points close to the diagonal indicate reliable probability estimates.
Support Vector Classifier (SVC) with probability=True uses Platt scaling internally,
which generally produces reasonably calibrated probabilities for binary classificati
on.
```

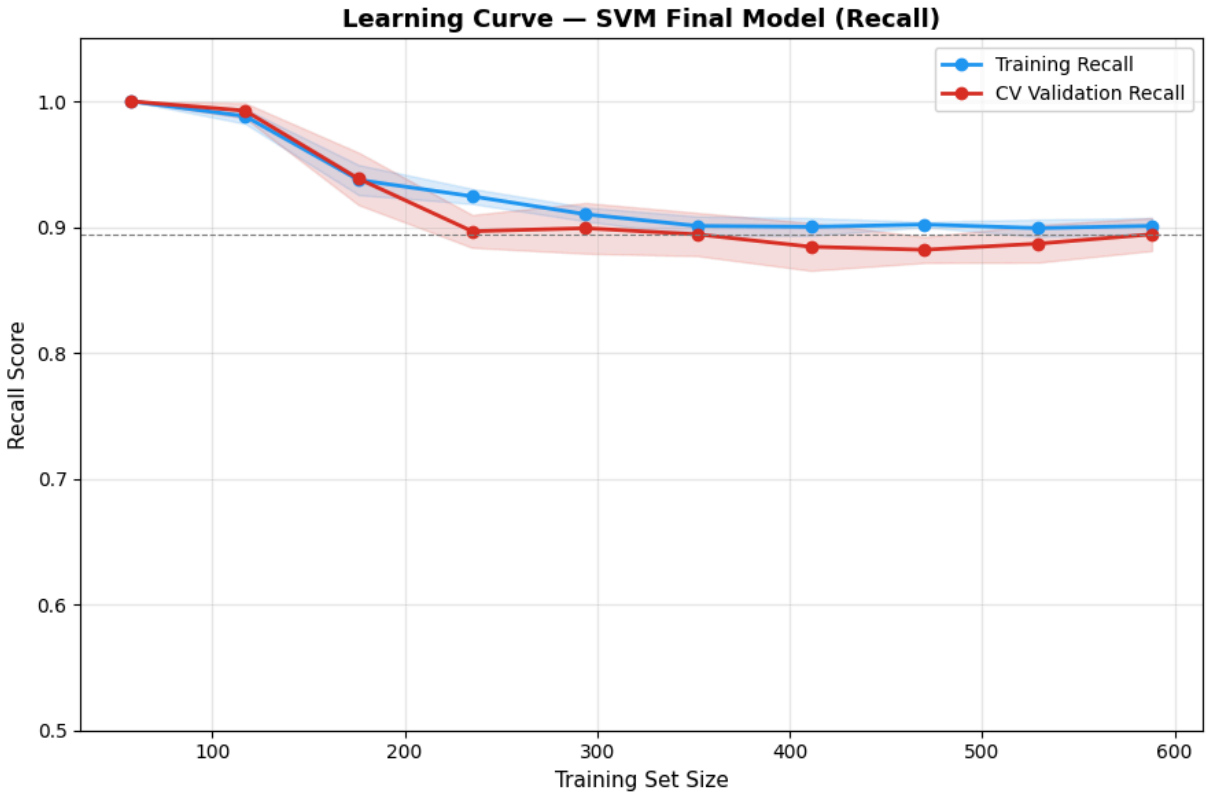
## BONUS — Learning Curve

A learning curve shows how model performance changes as we give it **more training data**.
It helps answer two important questions:

- Is the model still improving with more data — or has it plateaued?
- Is there a large gap between training and validation scores — suggesting overfitting?

If training score is high but validation score is low, the model is overfitting. If both scores are low, the model is underfitting. Ideally both curves converge at a high score — which is what we hope to see here.

In [25]:
```
from sklearn.model_selection import learning_curve

# Generate learning curve data using the best SVM pipeline
train_sizes, train_scores, val_scores = learning_curve(
    best_svm,
    X_train_top,
    y_train,
    cv=5,
    scoring='recall',
    train_sizes=np.linspace(0.1, 1.0, 10),
    n_jobs=-1
)

# Calculate mean and std
train_mean = train_scores.mean(axis=1)
train_std  = train_scores.std(axis=1)
val_mean   = val_scores.mean(axis=1)
val_std    = val_scores.std(axis=1)

fig, ax = plt.subplots(figsize=(9, 6))

# Training score
ax.plot(train_sizes, train_mean, 'o-', color='#2196F3',
        linewidth=2, markersize=6, label='Training Recall')
ax.fill_between(train_sizes, train_mean - train_std,
                train_mean + train_std, alpha=0.15, color='#2196F3')

# Validation score
ax.plot(train_sizes, val_mean, 'o-', color='#d73027',
        linewidth=2, markersize=6, label='CV Validation Recall')
ax.fill_between(train_sizes, val_mean - val_std,
                val_mean + val_std, alpha=0.15, color='#d73027')

ax.set_xlabel('Training Set Size', fontsize=11)
ax.set_ylabel('Recall Score', fontsize=11)
ax.set_title('Learning Curve — SVM Final Model (Recall)', fontsize=13, fontweight='
ax.legend(fontsize=10)
ax.set_ylim(0.5, 1.05)
ax.grid(True, alpha=0.3)
ax.axhline(y=val_mean[-1], color='grey', linestyle='--', linewidth=0.8)
plt.tight_layout()
plt.show()

print(f"""
Learning Curve Interpretation:
Final Training Recall:   {train_mean[-1]:.4f}
Final Validation Recall: {val_mean[-1]:.4f}
Gap:                     {train_mean[-1] - val_mean[-1]:.4f}

A small gap between training and validation recall indicates the model
generalizes well and is not significantly overfitting to the training data.""")
```




```
Learning Curve Interpretation:
Final Training Recall:   0.9010
Final Validation Recall: 0.8943
Gap:                     0.0068

A small gap between training and validation recall indicates the model
generalizes well and is not significantly overfitting to the training data.
```

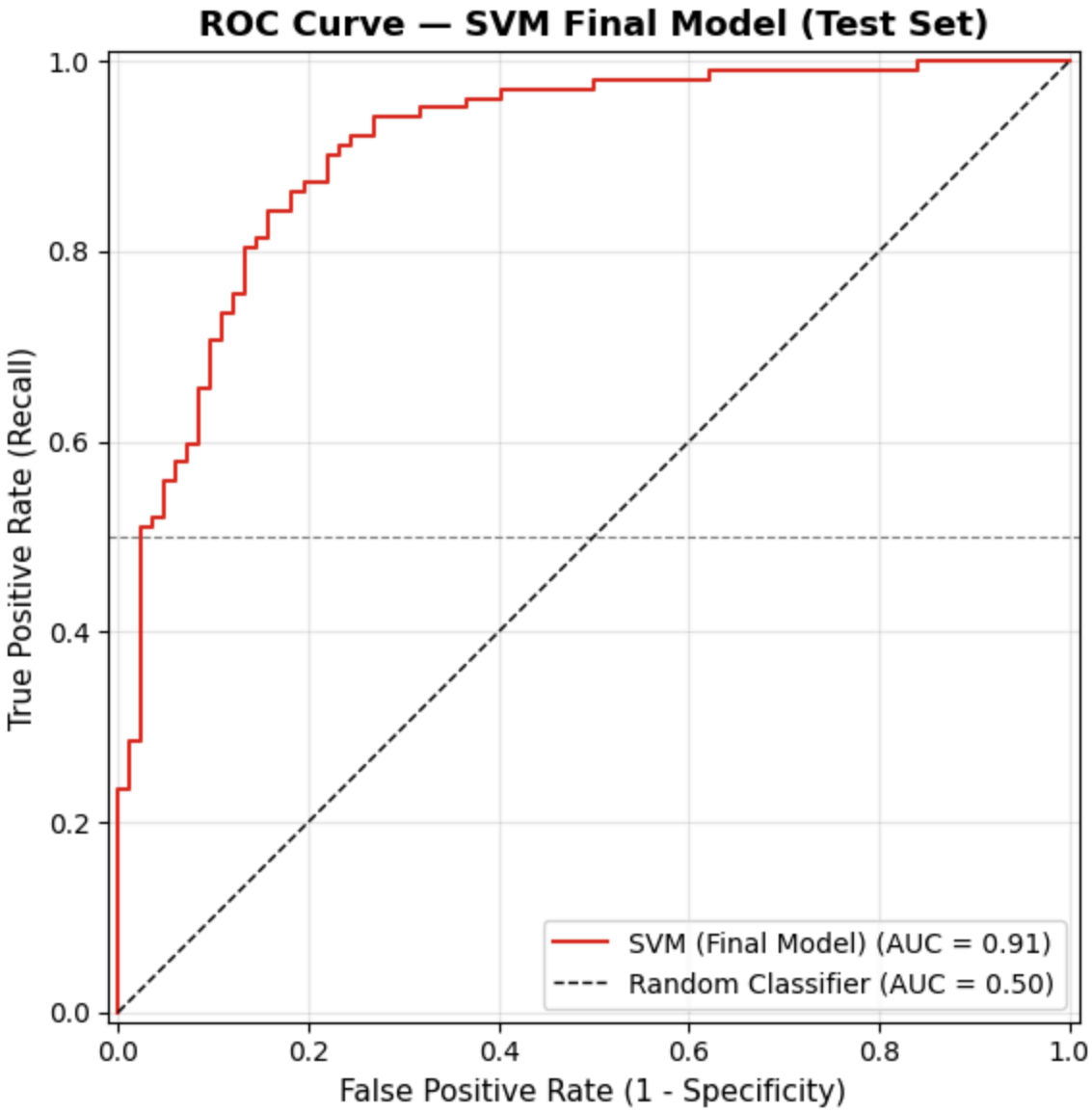
## BONUS — ROC Curve (Classification Equivalent of Residual Analysis)

In regression models, residual curves reveal how far predictions stray from actual values across the range of outputs. Classification models don't produce continuous predictions — they output class labels and probabilities — so a direct residual plot isn't applicable. The ROC curve serves as the closest equivalent.

Instead of asking "how far off was the prediction?" like a residual curve does, the ROC curve asks "at every possible decision threshold, how well does the model separate Disease from No Disease?" It captures the same underlying idea — how wrong can this model be, and under what conditions — just framed for a binary classification context.

The curve plots True Positive Rate (Recall) against False Positive Rate at every threshold. A perfect classifier hugs the top-left corner. A random classifier follows the diagonal. The area under the curve (AUC) summarizes the model's ability to correctly rank a Disease patient above a No Disease patient across all thresholds — our SVM achieved **0.91**, well above the **0.50** random baseline.

In [26]:
```
from sklearn.metrics import RocCurveDisplay

fig, ax = plt.subplots(figsize=(7, 6))

# Plot SVM ROC curve
RocCurveDisplay.from_predictions(
    y_test,
    y_proba_final,
    name='SVM (Final Model)',
    color='#d73027',
    ax=ax
)

# Add all four models' CV ROC-AUC scores as reference
ax.axhline(y=0.5, color='grey', linestyle='--', linewidth=0.8)
ax.plot([0, 1], [0, 1], 'k--', linewidth=1, label='Random Classifier (AUC = 0.50)')

ax.set_title('ROC Curve – SVM Final Model (Test Set)', fontsize=13, fontweight='bol
ax.set_xlabel('False Positive Rate (1 - Specificity)', fontsize=11)
ax.set_ylabel('True Positive Rate (Recall)', fontsize=11)
ax.legend(fontsize=10)
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"""
ROC Curve Interpretation:
The curve plots True Positive Rate (Recall) vs False Positive Rate at every
possible classification threshold.

A perfect classifier hugs the top-left corner (TPR=1, FPR=0).
A random classifier follows the diagonal dashed line (AUC=0.50).

Our SVM achieved AUC = {roc_auc:.4f} – meaning it correctly ranks a
randomly chosen Disease patient above a randomly chosen No Disease
patient {roc_auc*100:.1f}% of the time.
""")
```




```
ROC Curve Interpretation:
The curve plots True Positive Rate (Recall) vs False Positive Rate at every
possible classification threshold.

A perfect classifier hugs the top-left corner (TPR=1, FPR=0).
A random classifier follows the diagonal dashed line (AUC=0.50).

Our SVM achieved AUC = 0.9091 — meaning it correctly ranks a
randomly chosen Disease patient above a randomly chosen No Disease
patient 90.9% of the time.
```

## Final Evaluation: Interpretation & Reflection

The SVM model was selected based on cross-validation performance and evaluated on the hold-out test set for the first and only time.

**The model generalized well.** Final test performance exceeded CV estimates across all three key metrics — recall improved from **0.8943** to **0.9412**, ROC-AUC from **0.8768** to **0.9091**,

and accuracy from **0.8138** to **0.8424**. This tells us the model did not overfit to the training data.

**Recall was our priority — and the model delivered.** Correctly identifying **94%** of at-risk patients directly supports the goal of early detection and clinical decision-making. A missed diagnosis carries a far greater cost than a false alarm in this context.

**Notable tradeoff.** No Disease recall sits at **0.72**, meaning roughly **28%** of healthy patients are flagged as at-risk. This is an expected and acceptable tradeoff given that recall on the Disease class was our primary objective.

**Limitations.** Converting the target variable to binary removes information about disease severity. Features with high missing value rates such as **ca** and **thal** were imputed using mode, which may have introduced some bias. Another limitation is the feature selection step. Feature selection was performed before cross-validation which may introduce some leakage into our fold validation introducing some optimism. Feature selection was performed on all the training data instead of in folds causing peeking into the validation set. Future work could explore additional model architectures to push performance further.

## Conclusion

This project demonstrated the strength of machine learning models in effectively predicting heart disease using clinical features. SVM emerged as the best performing model for our chosen metric based on cross-validation and the test-set evalutation. This highlights the importance of understanding the underlying relationships and interactions in data that must be accounted for, especially within healthcare applications.

Overall, this analysis enforces that careful preprocessing, model selection, and evaluation is a necessity in producing reliable predictive models. Further model exploration could incorporate further feauture engineering, alternative imputation strategies, and more advanced ensemble methods.